Annual Report

Grant No. NAG-1-260

THE IMPLEMENTATION AND USE OF ADA ON DISTRIBUTED SYSTEMS WITH HIGH RELIABILITY REQUIREMENTS

Submitted to:

National Aeronautics and Space Administration
Langley Research Center
Hampton, Virginia 23665

Attention: Mr. Edmond H. Senn
ACD, MS 125

Submitted by:

J. C. Knight
Associate Professor

Report No. UVA/528213/CS86/108

March 1986

SCHOOL OF ENGINEERING AND APPLIED SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

UNIVERSITY OF VIRGINIA
CHARLOTTESVILLE, VIRGINIA 22901

Annual Report

Grant No. NAG-1-260

THE IMPLEMENTATION AND USE OF ADA ON DISTRIBUTED SYSTEMS WITH HIGH RELIABILITY REQUIREMENTS

Submitted to:

National Aeronautics and Space Administration
Langley Research Center
Hampton, Virginia 23665

Attention: Mr. Edmond H. Senn
ACD, MS 125

Submitted by:

J. C. Knight
Associate Professor

Department of Computer Science
SCHOOL OF ENGINEERING AND APPLIED SCIENCE
UNIVERSITY OF VIRGINIA
CHARLOTTESVILLE, VIRGINIA

Report No. UVA/528213/CS86/108

March 1986

Copy No. _____

Annual Report

Grant No. NAG-1-260

THE IMPLEMENTATION AND USE OF ADA ON DISTRIBUTED SYSTEMS WITH HIGH RELIABILITY REQUIREMENTS

Submitted to:

National Aeronautics and Space Administration
Langley Research Center
Hampton, Virginia 23665

Attention: Mr. Edmund H. Senn
ACD, MS 125

Submitted by:

J. C. Knight
Associate Professor

Report No. UVA/528213/CS86/108

March 1986

DEPARTMENT OF COMPUTER SCIENCE

Annual Report

Grant No. NAG-1-260

THE IMPLEMENTATION AND USE OF ADA ON DISTRIBUTED SYSTEMS WITH HIGH RELIABILITY REQUIREMENTS

Submitted to:

National Aeronautics and Space Administration
Langley Research Center
Hampton, Virginia 23665

Attention: Mr. Edmund H. Senn
ACD, MS 125

Submitted by:

J. C. Knight
Associate Professor

Department of Computer Science
SCHOOL OF ENGINEERING AND APPLIED SCIENCE
UNIVERSITY OF VIRGINIA
CHARLOTTESVILLE, VIRGINIA

Report No. UVA/528213/CS86/108

March 1986

Copy No. _____

CONTENTS

1. Introduction

The purpose of this grant is to investigate the use and implementation of Ada* in distributed environments in which reliability is the primary concern. In particular, we are concerned with the possibility that a distributed system may be programmed entirely in Ada so that the individual tasks of the system are unconcerned with which processors they are executing on, and that failures may occur in the software or underlying hardware.

Over the next decade, it is expected that many aerospace systems will use Ada as the primary implementation language. This is a logical choice because the language has been designed for embedded systems. Also, Ada has received such great care in its design and implementation that it is unlikely that there will be any practical alternative in selecting a programming language for embedded software.

The reduced cost of computer hardware and the expected advantages of distributed processing (for example, increased reliability through redundancy and greater flexibility) indicate that many aerospace computer systems will be distributed. The use of Ada and distributed systems seems like a good combination for advanced aerospace embedded systems.

During this grant reporting period our primary activities have been:

(1) Continued development and testing of our fault-tolerant Ada testbed.
(2) Development of suggested changes to Ada so that it might more easily cope with the failures of interest.

* Ada is a trademark of the U.S. Department of Defense

(3) Design of new approaches to fault-tolerant software in real-time systems, and the integration of these ideas into Ada.
(4) The preparation of various papers and presentations.

The various implementation activities of our fault-tolerant Ada testbed are described in section 2. During this grant reporting period we have made various reports about this work. Our activities in this area are described in section 3.

Most of the technical results of this grant have been documented in technical papers. A list of papers and reports prepared under this grant since it began, other than the annual and semiannual progress reports, is presented in appendix 1. Papers prepared and presented under this grant during the grant reporting period are included in this report as other appendices. A paper that was presented at the Fifteenth International Symposium on Fault-Tolerant Computing (FTCS15) is included as appendix 2. This paper is about backward error recovery in concurrent systems, and proposes a completely new approach to the subject. A paper that has been submitted to the Sixteenth International Symposium on Fault-Tolerant Computing is included as appendix 3. This paper extends the work descibed in the FTCS15 paper and introduces many new problems with backward error recovery that have not been described previously. We have not been informed of the status of that paper.

We have submitted previously a detailed paper describing our work on Ada to the IEEE Transactions on Software Engineering. We were informed that the paper would be accepted if we made several changes recommended by the referees. These changes were not substantial but we took the opportunity to revise the paper to reflect our latest views. This has resulted

in a new paper that will appear in the IEEE Transactions on Software Engineering in the near future. A copy of this latest paper is included as appendix 4. This paper is quite different from previous versions with the same title that have been supplied in previous grant reports.

2. Implementation Status

At the beginning of this grant reporting period a major activity that we undertook was to move the entire testbed to a network of Apollo DN300 workstations connected via a local area network. This was a substantial undertaking since the Apollo and the VAX systems are quite different. We set as a goal the development of a single version of the source text of the testbed which would reside on the VAX. The Apollo version would be build from this by various filters and other modification programs. The benefit of this approach is that a single system exists (in principle) and so bugs can be fixed and other changes made in one place but take effect on both computer systems. We achieved this goal and have ported successfully the entire testbed (but not the translator) to the Apollo system. The simple test cases that we have executed on the VAX implementation operate correctly on the Apollos.

One aspect of the translation of the system to the Apollos that surprised us was the Apollo system's communication performance. The VAX version of the testbed implements a virtual network using UNIX "pipes". Clearly since the Apollo implementation uses real processors and a real communication system, it was necessary to replace this part of the testbed. This was anticipated and performed. The Apollo communication system is a proprietary token-ring bus operating at 10 Mhz. Unfortunately this network is not directly available to an application program. We have been promised direct access on a restricted basis by Apollo Corporation on many occasions but this access has not been forthcoming.

The interface provided by the Apollo operating system is message based but requires one of the network nodes to be executing a message-switching program, thus implementing a "star" network. Further, all communication between nodes in this logical star is written to a disk and read from that disk on its way from one node to another. This slows the effective communication rate down by several orders of magnitude. Apart from the difficulty that we experienced in determining exactly how this interface worked and how to use it, when we finally got the necessary communication operating, we discovered transmission speeds that are of the same order as RS232.

During this grant reporting period there have been numerous changes to the computer systems that we use that have affected our Ada testbed. They are:

(1) The version of UNIX used by our VAX was changed from 4.1BSD to 4.2BSD. The testbed makes extensive use of the tasking features of UNIX and other system facilities for terminal access and control. Much of this interface was changed in the transition to 4.2BSD and the testbed was not operational initially under 4.2BSD. All the necessary changes have now been made and the testbed now operates correctly under 4.2BSD.

(2) The disk system for the Apollo network was enhanced substantially. Several new, small disk units were added to improve performance of the various Apollo workstations. In making this enhancement, subtle changes to the file system (normally transparent) were made and this affected the testbed in a number of ways. All the modifications needed

to cope with the changes have now been made in the testbed.

(3) The operating system used by the Apollo network was upgraded to take advantage of new facilities offered by Apollo. This has had some subtle effects on the services that the testbed uses and stopped the testbed from operating on the Apollos for some time. We have made the necessary changes to the testbed to cope with the operating system changes.

Given the poor communication performance offered by the Apollo network we question the utility of having the testbed available on that network. Our department expects to receive equipment to implement a new network from a different supplier and we are considering using the new equipment rather than continuing to use the Apollos.

We have very little control over the poor communications facilities provided to us despite the fact that they play a significant role in the testbed's overall performance. However, in testing the testbed, we have discovered that it is extremely inefficient in its own right. We never intended the testbed to be an efficient implementation since our primary concern was functionality. However, in trying to run test programs, we have been frustrated by the slowness of the implementation.

During this grant reporting period we have extended the translator for the subset of Ada that interests us and the translator is now essentially complete. Although it operates on the VAX, it must be kept in mind that it generates code for a synthetic Ada machine and so its output can be used by the testbed when operating on any target, in particular the Apollo network. The code quality produced by the translator is quite poor since,

once again, efficiency is not a concern of this project.

We had expected to spend some effort on improving the efficiency of our testbed implementation during the grant reporting period. However, we limited our effort to making extensive revisions to the source code of the testbed and translator in order to make both more easily maintained. This will aid any effort to improve efficiency should we undertake this in the future. A major change was made to the translator. In its original implementation, there were two passes, one was written in Pascal and the second in C. The entire second pass has now been rewritten in Pascal and streamlined.

3. Professional Activities

During this grant reporting period we gave a seminar describing the work at NASA's Goddard Space Flight Center. We also attended a workshop at the University of Houston at Clearlake, and made an extensive presentation at the workshop on our activities.

Appendix 1

Papers and reports.

Papers And Reports

The following is a list of papers and reports, other than progess reports, prepared under this grant.

(1) Knight, J.C. and J.I.A. Urquhart, "Fault-Tolerant Distributed Systems Using Ada", Proceedings of the *AIAA Computers in Aerospace Conference*, October 1983, Hartford, CT.

(2) Knight, J.C. and J.I.A. Urquhart, "The Implementation And Use Of Ada On Fault-Tolerant Distributed Systems", *Ada LETTERS*, Vol. 4 No. 3 November 1984.

(3) Knight, J.C. and J.I.A. Urquhart, "On The Implementation and Use of Ada on Fault-Tolerant Distributed Systems", *IEEE Transactions on Software Engineering*. to appear.

(4) Knight J.C. and S.T. Gregory, "A Testbed for Evaluating Fault-Tolerant Distributed Systems", Digest of Papers FTCS-14: *Fourteenth Annual Symposium on Fault-Tolerant Computing*, June 1984, Orlando, FL.

(5) Knight J.C. and S.T. Gregory, "A New Linguistic Approach To Backward Error Recovery", Digest of Papers FTCS-15: *Fifteenth Annual Symposium on Fault-Tolerant Computing*, June 1985, Ann Arbor, MI.

(6) Knight J.C. and S.T. Gregory, "The Provision of Backward Error Recovery in Realistic Programming Languages", submitted to FTCS-16: *Sixteenth Annual Symposium on Fault-Tolerant Computing*, July 1986, Vienna, Austria.

(7) Knight, J.C. and J.I.A. Urquhart, "Difficuties With Ada As A Language For Reliable Distributed Processing", Unpublished.

(8) Knight, J.C. and J.I.A. Urquhart, "Programming Language Requirements For Distributed Real-Time Systems Which Tolerate Processor Failure", Unpublished.

Appendix 2

A New Linguistic Approach

To Backward Error Recovery

Presented At

The Fifteenth International Symposium On Fault-Tolerant Computing

Ann Arbor, Michigan

A NEW LINGUISTIC APPROACH TO BACKWARD ERROR RECOVERY*

Samuel T. Gregory John C. Knight

Department of Computer Science
University of Virginia
Charlottesville, Virginia, U.S.A. 22903
(804) 924-7605

ABSTRACT

Issues involved in language facilities for backward error recovery in critical, real-time systems are examined. Previous proposals are found lacking. The dialog, a new building block for concurrent programs, and the colloquy, a new backward error recovery primitive, are introduced to remedy the situation. The previous proposals are shown to be special cases of the colloquy. Thus, the colloquy provides a general framework for describing backward error recovery in concurrent programs.

Subject Index:
Reliable Software - Interprocess Communication and Synchronization

*This work was sponsored by NASA grant number NAG1-260 and has been cleared for publication by the sponsoring organization.

1. INTRODUCTION

In this paper we examine the issues involved in the use of backward error recovery in critical, real-time systems. In particular, we are concerned with language facilities that allow programmers to specify how alternate algorithms are to be applied in the event that an error is detected. The best-known approach is the *conversation*[1]. Many difficulties with conversations have been pointed out including the lack of any time-out provision and the possibility of deserter processes. We introduce a new building block for concurrent programs called the *dialog* and a new backward-error-recovery primitive called the *colloquy* that remedy the various limitations of the conversation. The colloquy is constructed from dialogs and provides a general framework for describing backward error recovery in concurrent programs.

All of the syntactic proposals that we introduce are derived from Ada® [2]. The dialog and colloquy are proposed as general concepts but the specific syntax for their use is given as extensions to Ada. The actual syntax is irrelevant; the concepts could be used in many other programming languages. However, once chosen, a rigid syntax can allow a compiler to enforce certain of the semantic rules.

In section two, we briefly describe the concept of the conversation and the associated syntactic proposals that have been made. Issues that have been raised with conversations are discussed in section three. In section four, we present a syntax for the dialog called the *discuss* statement. In section five, we introduce the colloquy and a new statement called the *dialog_sequence* which allows the specification of the actions needed for a colloquy. In section six, we discuss the use of colloquys in the implementation of all previous approaches to backward error recovery.

®Ada is a registered trademark of the U.S. Government (Ada Joint Program Office).

2. CONVERSATIONS

The *conversation* is the canonical software fault-tolerance proposal for dealing with communicating processes. In a conversation a group of processes separately establish recovery points and begin communicating. At the end of their communication (i.e. the end of the conversation), which may include the passage of multiple distinct sets of information, they each wait for the others to arrive at an acceptance test for the group. If they pass the acceptance test, they *commit* to the information exchange that has transpired by discarding their recovery points and proceeding. Should they fail the acceptance test, they all restore their states from the recovery points. No process is allowed to *smuggle* information in or out by communicating with a process that is not participating in the conversation. Conversations can be nested; from the point of view of a surrounding conversation, a nested conversation is an atomic action[3].

Although not explicitly stated in the literature, it is assumed that if an error occurs during a conversation such that the acceptance test fails, the *same* set of conversant processes attempt to communicate again once individually rolled back and reconfigured (rather than proceeding on unrelated activities). It follows that they eventually reach the *same* acceptance test again. It is also presumed that any other failure of one of the processes is taken as equivalent to a failure of the acceptance test by all of them.

The processes in a conversation are the components of a system of processes. Error detection mechanisms for this system consist of announcement of failure by any one of the components and the single acceptance test. The acceptance test evaluates the combined states of the component processes with the designed intent of their communications. Damage assessment is complete *before* execution begins since the individual states of all the processes involved in the conversation are suspect.

but no other processes are affected. Error recovery consists of restoring each process to the state it had as it entered the conversation, and the system of processes continues with its service by allowing each process to re-try the communication perhaps using an alternate mechanism within that process for the communication activity.

Conversations were originally proposed as a structuring or design concept without any syntax that might allow enforcement of the rules. Russell[4] has proposed the "Name-Linked Recovery Block" as a syntax for conversations. The syntax appropriates that of the recovery block[5]. What would otherwise be a recovery block, becomes part of a conversation designated by a conversation identifier. The primary and alternate activities of the recovery block become that process' primary and alternate activities during the conversation, and the recovery block's acceptance test becomes that portion of the conversation's acceptance test appropriate to this process. The conversation's acceptance test is evaluated after the last conversant reaches the end of its primary or alternate. If any of the processes fail its acceptance test, all conversants are rolled back.

Kim has examined several more possible syntaxes for conversations[6]. His approaches assume the use of monitors[7] as the method of communication among processes. He examines the situation from two philosophies toward grouping. In one scheme, the conversing activities are grouped with their respective processes' source code, but are well marked at those locations. In another scheme, the conversing actions of the several processes are grouped into one place so that the conversation has a single location in the source code. The issue he is addressing is whether it is better to group the text of a conversation and scatter the text of a process or to group the text of a process and scatter the text of a conversation. A third scheme attempts to resolve the differences between the first two.

3. ISSUES WITH CONVERSATIONS

Desertion is the failure of a process to enter a conversation or arrive at the acceptance test when other processes expect its presence. Whether the process will never enter the conversation, is simply late, or enters the conversation only to take too long or never arrive at the acceptance test, does not matter to the others if they have real-time deadlines to meet. Each process may have its own view of how long it is willing to wait, especially since processes may enter a conversation asynchronously. Whether they protect inter-process communications or sequential parts of processes, acceptance tests must be reached and reached on time for the results to be useful. Meeting real-time deadlines is as important to providing the specified service as is producing correct output. In order to deal effectively with desertion, especially in critical systems, some form of timing specification on communication and on sequential codes is vital.

When it needs to communicate, a process enters a conversation and stays there, perhaps through many alternate algorithms, until the communication is completed successfully. The same group of processes are required to be in the alternate interactions as were in the primary. The recovery action merely sets up the communication situation again. In the original form of conversation, once a process enters the construct, it cannot break out and *must* continue trying with the same set of other processes, including one or more which may be incapable of correct operation. In practice, when a process fails in a primary attempt at communication with *one group of processes* to achieve its goal, it may want to attempt to communicate with an *entirely different group* as an alternate strategy for achieving that goal; in fact, different processes might make different numbers of attempts at communicating. Conversations do not allow this, although it is not desertion if it is systematic and intended.

In a conversation, once individually rolled back and reconfigured, the same set of conversant processes attempt to communicate again, and eventually reach the *same* acceptance test again. True *independence* of algorithms between primary and alternates, within the context of backward error recovery, might require very *different* acceptance tests for each algorithm, particularly if some of them provide significantly degraded services. A single test for achievement of a process' goal at a particular point in its text would of necessity have to be general enough to pass results of the most degraded algorithm. This might be too general to enable it to catch errors produced by other, more strict, algorithms. These considerations suggest the need for separate acceptance tests specifically tailored for each of the primary and alternate algorithms.

It must also be remembered, that although each process has its own reasons for participating, there is a goal for the *group* of processes as well. Rather than combine the individual goals of the many participants with the group goal in a single acceptance test (perhaps allowing the programmer to forget some), and rather than replicating the test for achievement of the group goal within every participant, there should be a separate acceptance test for each participant and another for the group.

A final problem with the conversation concept as it was originally defined, is that if a process runs out of alternates, no scheme is provided or mentioned for dealing with the situation.

4. THE DIALOG

We define a *dialog* to be an occurrence in which a set of processes:

(a) establish individual recovery points,

(b) communicate among themselves and with no others,

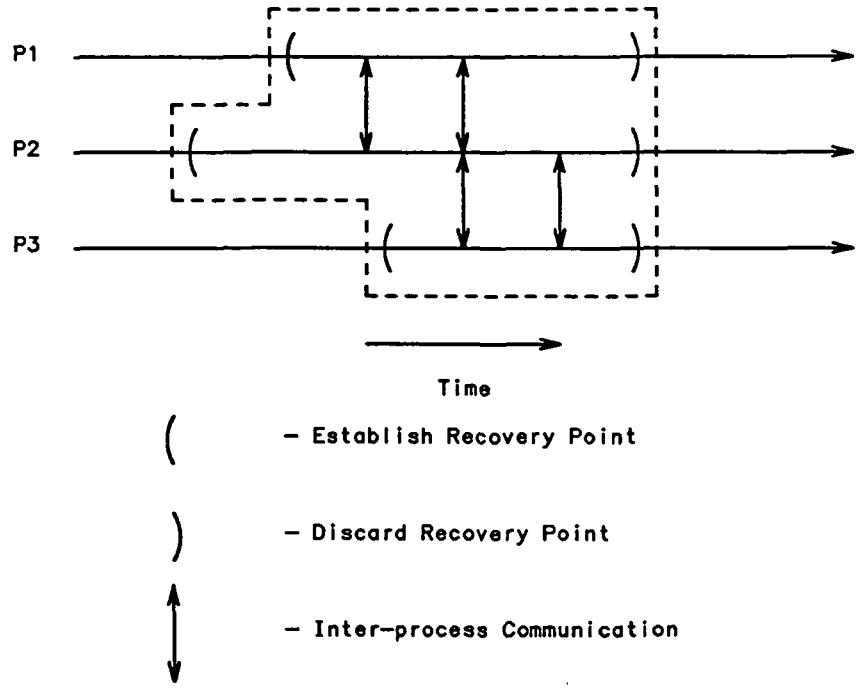
(c) determine whether all should discard their recovery points and proceed or restore their states from their recovery points and proceed, and

(d) follow this determination.

Success of a dialog is the determination that all participating processes should discard their recovery points and proceed. *Failure* of a dialog is the determination that they should restore their states from their recovery points and proceed. Nothing is said about what should happen *after* success or failure; in either case the dialog is complete. Dialogs may be properly nested, in which case the set of processes participating in an inner dialog is a subset of those participating in the outer dialog. Success or failure of an inner dialog does not necessarily imply success or failure of the outer dialog. Figure 1 shows a set of three processes communicating within a dialog.

We introduce the *discuss* statement as a syntactic form that can be used to denote a dialog. Figure 2 shows the general form of a discuss statement. The *dialog_name* associates a particular discuss statement with the discuss statements of the other processes participating in this dialog, *dynamically* determining the constituents of the dialog. This association cannot in general be known statically. At execution time, when control enters a process' discuss statement with a given dialog name, that process becomes a participant in a dialog. Other participants are any other processes which have already likewise entered discuss statements with the same dialog name and have not yet left, and any other processes which enter discuss statements with the same dialog name before this process leaves the dialog. Either all participants in a dialog leave it with their respective discuss statements successful, or all leave with them failed, i.e. the dialog succeeds or fails.

The *sequence of statements* in the discuss statement represent the actions which are this process' part of the group's actions within their dialog. Any inter-process



Three Processes Communicating in a Dialog
Figure 1

```
DISCUSS dialog_name BY
    sequence_of_statements
TO ARRANGE Boolean_expression;
```

A DISCUSS Statement
Figure 2

communication *must take place within* this sequence of statements (i.e. be protected by a dialog). The discuss statement fails if an exception is raised within it, if an enclosed *dialog_sequence* (see below) fails, or if any timing constraint is violated.

The *Boolean_expression* is an acceptance test on the results of executing the sequence of statements. It represents the process' *local* goal for the interactions in the dialog. It is evaluated after execution of the sequence of statements. If this Boolean expression or that in the corresponding discuss statement of any other process participating in this dialog is evaluated **false**, the discuss statement of each participant in the dialog *fails*. If all of the local acceptance tests succeed, the common goal of the group, i.e. the *global* acceptance test is evaluated. If this common goal is **true**, the corresponding discuss statements of all participants in the dialog succeed; otherwise they fail. Syntactically, the common goal is specified by a parameterless Boolean function with the same name as the dialog name in the discuss statement.

We stated that the participants in a particular dialog cannot be known statically. There may be, say, three processes whose texts contain references to a particular dialog name. If two of them enter a dialog using that name, questions might arise about participation of the third. The third process may be executing some other portion of its code so that it is unlikely to enter a dialog of that name in the near future. If the two processes reach and pass their acceptance tests, they, being the only participants in the dialog, can leave it -- the third process is not necessary to the dialog, so is not a deserter. If the dialog fails due to an acceptance test or a timeout (see below), the problem is not guaranteed to be the absence of the third process, so again it is not (necessarily) a deserter. If the dialog has no time limit specified (see below), that had to be by conscious effort of the programmer, so the two processes becoming "hung" in the dialog while waiting for the third was *not* unplanned.

The dialog names used in discuss statements are required to be declared in *dialog declarations*. The general form of a dialog declaration is:

```
DIALOG function_name SHARES ( name_list );
```

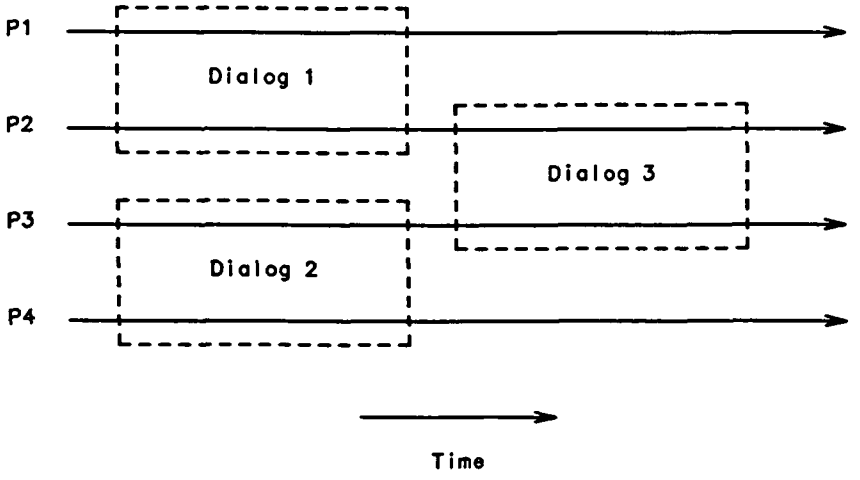
The *function_name* is the identifier being declared as a dialog name (and the name of the function defining the global acceptance test). The names mentioned in the *name_list* are the names of *shared* variables which will be used within dialogs that use this dialog name. This includes variables used within the function that implements the global acceptance test. Only a variable so named may be used within a discuss statement, and then only within discuss statements using a dialog name with that variable's name in its dialog declaration. The significance of these rules is that the set of shared variables can be locked by the compiler and execution-time support system to prevent smuggling. In effect, the actions of the dialog's participants are made to appear atomic to other processes with respect to these variables. (Our implementation, not described here, also prevents smuggling via messages or rendezvous).

The Boolean function named by the dialog name is evaluated after all processes in the dialog have evaluated their respective Boolean expressions and they all evaluate to **true**. It is only evaluated once for an instance of the dialog; i.e. it is not evaluated by each participating process. Thus no process can leave a dialog until all processes currently in that dialog leave with the same success, and success involves the execution of both a local and a global acceptance test.

5. THE COLLOQUY

A *colloquy* is a semantic construct that solves the problems of conversations. Unlike conversations, the rules of order and participation are well-defined and explicitly laid out.

A colloquy is a collection of dialogs. At execution time, a dialog is an interaction among processes. Each individual process has its own *local* goal for participating in a dialog, but the group has a larger *global* goal; usually providing some part of the service required of the entire system. If, for whatever reason, any of the local goals or the global goal is not achieved, a backward error recovery strategy calls for the actions of the particular dialog to be undone. In attempting to ensure continued service from the system, each process may make another attempt at achieving its original local goal, or some *modified* local goal through entry into a *different* dialog. Each of the former participants of the now defunct dialog may choose to interact with an *entirely separate* group of processes for its alternate algorithm. The altered constituency of the new dialog(s) most certainly requires new statement(s) of the original global goal. The set of dialogs which take place during these efforts on the processes' part is a *colloquy*. A set of four processes engaged in a colloquy that involves three dialogs is shown in Figure 3.



Four Processes in a Colloquy of Three Dialogs
Figure 3

A colloquy, like a dialog or a rendezvous in Ada, does not exist syntactically but is entirely an execution-time concept. The places where the text of a process statically announces entry into colloquys are marked by a variant of the Ada select statement called a *dialog_sequence*.

The general form of a dialog_sequence is shown in Figure 4. At execution time, when control reaches the **select** keyword, a recovery point is established for that process. The process then *attempts* to perform the activities represented in Figure 4 by `attempt_1`. The attempt is actually a discuss statement followed by a sequence of statements. To ensure proper nesting of dialogs and colloquys, a discuss statement may appear only in this context. If the performance of these activities is *successful*, control continues with the statements following the dialog_sequence. The term "success" here means that no defensive, acceptability, or timing checks occurring within the attempt detected an error, and that no exceptions (if the language has exceptions) were propagated out to the attempt's discuss statement. If the attempt was not successful, the process' state is restored from the recovery point and the other attempts will be tried in order. Thus, the dialog_sequence enables the

```
SELECT
      attempt_1
   OR
      attempt_2
   OR
      attempt_3

   TIMEOUT simple_expression
      sequence _of_statements

   ELSE
      sequence _of_statements
END SELECT;
```

Dialog_Sequence
Figure 4

programmer to provide a primary and a list of alternate algorithms by which the process may achieve its goals at that locus of its text.

Exhaustion of all attempts with no success brings control to the **else** part after restoration of the process' state from the recovery point. The **else** part contains a sequence of statements which allows the programming of a "last ditch" algorithm for the process to achieve its goal. If this sequence of statements is successful, control continues after the dialog_sequence. If not, or if there was no statement sequence, the surrounding attempt fails.

Timing constraints can be imposed on colloquys (and hence on dialogs). Any participant in a colloquy can specify a timing constraint which consists of a simple expression on the **timeout** part of the dialog_sequence. Absence of a timing constraint must be made explicit by replacing the simple expression with the keyword **never**. A timing constraint specifies an interval during which the process may execute as many of the attempts as necessary to achieve success in one of them. Should an attempt achieve success or the list of attempts be exhausted without success before expiration of the interval, further actions are the same as for dialog_sequences without timing specifications. However, if the interval expires, the current attempt fails, the process' state is restored from the recovery point, and execution continues at the sequence of statements in the **timeout** part. The attempts of the other processes in the same dialog also fail but their subsequent actions are determined by their own dialog_sequences. If several participants in a particular colloquy have timing constraints, expiration of one has no effect on the other timing constraints. The various intervals expire in chronological order. As with the else part, the timeout part allows the programming of a "last ditch" algorithm for the process to achieve its goal, and is really a form of forward recovery since its effects will not be undone, at least at this level. If the sequence of statements in the

timeout part is successful, control continues after the dialog_sequence. If not, or if there were no statement sequence, the surrounding attempt fails.

In any attempt, a statement sequence (which is logically outside the dialog_sequence) can follow the discuss statement to provide specialized post-processing after the recovery point is discarded if the attempt succeeds. It is not subject to this dialog_sequence's timing constraint.

The programmer is reminded by its position after the **timeout** part that the **else** part is not protected by the timer, and that it is reached only after other (potentially time-consuming) activities have taken place. The structure of the dialog_sequence also requires no acceptance check on these activities. The implication of these two observations is that the last ditch activities need to be programmed very carefully.

A *fail* statement may occur only within a sequence of statements contained within a dialog_sequence. Execution of a fail statement causes the encompassing attempt to fail. The fail statement is intended for checking within an attempt. For example, it can be used to program explicit defensive checks on inputs such as:

```
IF input_variable < lower_bound THEN
   FAIL;
END IF;
```

It can also be used to simplify the logical paths out of an attempt should some internal case analysis reach an "impossible" path. With the fail statement, the programmer does not have to make the code for the attempt complicated by providing jumps or other paths to the acceptance test or to insure that some part of the test is always **false** for such a special path. The fail statement can also be used to provide sequences of statements for the **else** and **timeout** parts that make failure explicit rather than implicit (i.e. failure is indicated by their *absence*).

6. OTHER LANGUAGE FACILITIES

Dialog_sequences can be used to construct deadlines[8], generalized exception handlers[9], recovery blocks, traditional conversations, exchanges[10], and s-conversations[11]. Thus the colloquy is at least as powerful as each of these previously proposed constructs for provision of fault tolerance. For the sake of brevity, we will illustrate only the programming of a recovery block.

A recovery block is a special case of a colloquy in which there is only one process participating, every dialog uses the same acceptance test, there is no timing requirement, and there are no "last ditch" algorithms to prevent propagation of failures of the construct. Figure 5 shows a dialog_sequence that is equivalent to the recovery block shown in Figure 6. The use of the fail statement in the dialog_sequence makes explicit the propagation of the error to a surrounding context just as does the **else error** closing of the recovery block. In the dialog_sequence, the Boolean expression is repeated in the discuss statements rather than gathered into the dialog function because we want to be able to include local variables in it as a programmer of the recovery block would. Should an error be detected in `statement_sequence_1`, the state is restored and `statement_sequence_2` is executed, and so on. Finally, should an error be detected in `statement_sequence_3`, the state is restored and the error is signaled in a surrounding context. An error may be detected by evaluation of `boolean_expression_1` to **false**, or by violation of some underlying interface (such as raising of an exception).

7. CONCLUSIONS

We have introduced a new linguistic construct, the colloquy, which solves the problems identified in the earlier proposal, the conversation. We have shown that

```
FUNCTION abc RETURNS boolean IS BEGIN RETURN TRUE; END abc;
  ....
DIALOG abc SHARES ( );
  ....

SELECT
      DISCUSS abc BY
         statement_sequence_1
      TO ARRANGE boolean_expression_1;

   OR
      DISCUSS abc BY
         statement_sequence_2
      TO ARRANGE boolean_expression_1;

   OR
      DISCUSS abc BY
         statement_sequence_3
      TO ARRANGE boolean_expression_1;

   TIMEOUT NEVER;

   ELSE
       FAIL; — Omitting this line does not change the semantics.
END SELECT;
```

Specification of Colloquy for a Recovery Block
Figure 5

```
ENSURE boolean_expression_1 BY
   statement_sequence_1

ELSE BY
   statement_sequence_2

ELSE BY
   statement_sequence_3

ELSE ERROR;
```

A Recovery Block
Figure 6

the colloquy is at least as powerful as recovery blocks, but it is also as powerful as all the other language facilities proposed for other situations requiring backward

error recovery; recovery blocks, deadlines, generalized exception handlers, traditional conversations, s-conversations, and exchanges.

The major features that distinguish the colloquy are:

(1) The inclusion of explicit and general timing constraints. This allows processes to protect themselves against any difficulties in communication that might prevent them from meeting real-time deadlines. It also effectively deals with the problem of deserter processes.

(2) The use of a two-level acceptance test. This allows much more powerful error detection because it allows the tailoring of acceptance tests to specific needs.

(3) The reversal of the order of priority of alternate communication attempts and of recovery points. This allows processes to choose the participants in any alternate algorithms rather than being required to deal with a single set of processes.

(4) A complete and consistent syntax that is presented as extensions to Ada but could be modified and included in any suitable programming language.

Sample programs that have been written (but not executed) using the colloquy show that extensive backward error recovery can be included in these programs simply and elegantly. We are presently implementing these ideas in an experimental Ada testbed.

This paper is not a formal statement of these concepts. The reader may correctly feel that important detail has been omitted. We are only able to present informally the key concepts in a paper of this length. For more details, see [12].

8. ACKNOWLEDGEMENTS

This work was sponsored by NASA grant number NAG1-260 and has been cleared for publication by the sponsoring organization.

REFERENCES

(1) Randell B., "System Structure for Software Fault Tolerance," *IEEE Transactions on Software Engineering*, SE-1(2), pp. 220-232, June 1975.

(2) *Reference Manual for the Ada Programming Language*, ANSI/MIL-STD-1815A, 22 January 1983.

(3) Lomet D.B., "Process Structuring, Synchronization and Recovery Using Atomic Actions," *SIGPLAN Notices*, 12(3), pp. 128-137, March 1977.

(4) Russell D.L., M.J. Tiedeman, "Multiprocess Recovery Using Conversations," *Digest of Papers FTCS-9: Ninth Annual Symposium on Fault-Tolerant Computing*, p. 106, June 1979.

(5) Horning J.J., et al., "A Program Structure for Error Detection and Recovery," pp. 171-187 in *Lecture Notes in Computer Science* Vol. 16, ed. E. Gelenbe and C. Kaiser, Springer-Verlag, Berlin, 1974.

(6) Kim K.H., "Approaches to Mechanization of the Conversation Scheme Based on Monitors," *IEEE Transactions on Software Engineering*, SE-8(3), pp. 189-197, May 1982.

(7) Per Brinch Hansen, *The Architecture of Concurrent Programs*, Prentice-Hall, Englewood Cliffs, NJ, 1977.

(8) Campbell R.H., K.H. Horton, G.G. Belford, "Simulations of a Fault-Tolerant Deadline Mechanism," *Digest of Papers FTCS-9: Ninth Annual Symposium on Fault-Tolerant Computing*, pp. 95-101, 1979.

(9) Salzman E.J., *An Experiment in Producing Highly Reliable Software*, M.Sc. Dissertation, Computing Laboratory, University of Newcastle upon Tyne, 1978.

(10) Anderson T., J.C. Knight, "A Framework for Software Fault Tolerance in Real-Time Systems," *IEEE Transactions on Software Engineering*, **SE-9**(3), pp. 355-364, May 1983.

(11) Jalote P., R.H. Campbell, "Fault Tolerance Using Communicating Sequential Processes," *Digest of Papers FTCS-14: Fourteenth International Conference on Fault-Tolerant Computing*, pp. 347-352, 1984.

(12) Gregory S.T., *Programming Language Facilities for Comprehensive Software Fault-Tolerance in Distributed Real-Time Systems*, Ph.D. Dissertation, Department of Computer Science, University of Virginia, 1985.

Appendix 3

The Provision of Backward Error Recovery In Realistic Programming Languages

Submitted To

The Sixteenth International Symposium On Fault-Tolerant Computing

Vienna, Austria

THE PROVISION OF BACKWARD ERROR RECOVERY IN REALISTIC PROGRAMMING LANGUAGES*

Samuel T. Gregory John C. Knight**

Department of Computer Science
University of Virginia
Charlottesville, Virginia 22903, U.S.A.
(804) 924-7605

ABSTRACT

We have introduced previously the *colloquy*, a general linguistic approach to backward error recovery. We have shown that it solves all the problems that have been raised in the past with *conversations*. It also includes, as special cases, all previously proposed language structures for backward error recovery in both sequential and concurrent programs. In this paper we examine the problem of providing backward error recovery in *realistic* programming languages. By realistic we mean programming languages with sufficient expressive power that they can be used for substantial applications. We use Ada as an example. This examination reveals several new problems that have not been addressed previously. In revealing these problems, we do not condemn Ada. Rather we show the relative immaturity of the backward error recovery approach in relation to languages of which Ada is but one example. We show that the source of the problems is the continuous need to be able to define a *recovery line* so as to be able to perform state restoration. Many language constructs that have not been addressed by other researchers, such as shared data, process creation and destruction, and pointers, make the establishment of a recovery line extremely difficult. We present solutions to the problems identified.

Subject Index:
Software fault tolerance, backward error recovery, concurrent systems, Ada.

*This work was sponsored by NASA grant number NAG1-260 and has been cleared for publication by the authors' affiliation.

**Presenter at FTCS-16 if paper is accepted.

1. INTRODUCTION

In this paper we consider the problem of providing fault tolerance in *concurrent* systems through *backward error recovery*. In particular, we are concerned with programming language primitives that allow the specification of fault-tolerant concurrent systems, and the impact that such primitives have on other elements of modern programming languages.

The best-known approach to structuring concurrent systems to allow backward error recovery is the *conversation*[1]. There are many well-known difficulties with conversations (and similar approaches), and we have proposed an alternative mechanism, the *colloquy*[2], that has none of the previously-identified disadvantages of the conversation.

Subsequent to defining the colloquy, we have attempted to introduce the concept into *realistic* programming languages. By realistic we mean programming languages that contain the necessary facilities to program modern, large-scale applications. We have found that several new and potentially serious difficulties arise when this integration is attempted. The difficulties that we discuss arise because of an inherent *conflict* between the needs of the applications' programmer, as exemplified by modern language design, and the fundamental needs of backward error recovery in concurrent systems.

In this paper, we present these difficulties and show how they can be overcome by making extensive changes to realistic programming languages. All of the difficulties that we discuss use Ada® [3] as an example. In examining problems with including backward error recovery in Ada, we are *not* criticizing Ada. Rather, we show the *conflict* between backward error recovery and the semantics of realistic

®Ada is a registered trademark of the U.S. Government (Ada Joint Program Office).

programming languages using Ada merely as an example. The actual programming language is irrelevant, and we stress the fact that the problems raised *will* occur in many other programming languages.

Our discussion of this subject is arranged as follows. We review the concepts of backward error recovery, summarize the colloquy, and discuss the general issues that arise with concurrent systems in section two. In section three, the effects of backward error recovery on *program structure* are discussed. In section four, we consider the problems of *shared objects* in a system that has to provide backward error recovery, and in section five, we examine the effects of *dynamic process manipulation*. We present our conclusions in section six.

2. BACKWARD ERROR RECOVERY

Despite the use of all available techniques for fault avoidance and fault removal, experience has shown that production software contains faults. Fault tolerance is an approach to improving the reliability of software by tolerating the effects of faults and continuing to provide service.

One method of building fault-tolerant software is backward error recovery. An erroneous state, once detected, is replaced by a state, assumed not to be erroneous, that existed at some prior time and had been saved. Service is continued by attempting to restart execution using the restored state and an alternate algorithm. Backward error recovery has been shown to be effective in improving reliability in an empirical study[4].

For sequential programs, backward error recovery has been the subject of extensive study[5, 6, 7, 8, 9, 10]. The general concept requires that a program signal

during execution when it reaches a point, termed a *recovery point*, to which it might need to return. The state of the computation is then saved, at least conceptually, and the program proceeds. After performing some further computation, the program checks its current state by evaluating a Boolean condition, termed an *acceptance test*. If the Boolean condition is satisfied, the state saved at the recovery point is discarded. If the condition is not satisfied, the state is restored to that which was saved (a process known as *recovery*), and the program proceeds with some alternate computation. This technique is referred to as backward error recovery since the restored state is one that was saved in the past. The part of the program's execution from the establishment of the recovery point until it it discarded is termed a *recovery region*.

The best-known linguistic structure designed to support backward error recovery in sequential programs is the *recovery block*[5, 6, 7, 8, 9, 10]. Hardware assistance in the form of the recovery cache[11] is advocated for implementations of the recovery block to provide efficient state saving and state restoration.

Extensions of backward error recovery to concurrent programs has proved quite difficult. A well-known source of difficulty is that a set of processes must agree about establishing and discarding their individual recovery points. Usually, it is not possible for a single process to establish its own recovery point and attempt isolated recovery when an erroneous state is detected. If a process has been involved in *any* communication, then it will be necessary for the processes with which it communicated to recover also. For backward error recovery to work, it is essential that the states of the *entire* set of communicating processes be restored to a previous value; this includes "undoing" any communication that took place. If the various recovery points are not coordinated, any attempt to establish a consistent state can lead to uncontrolled roll back; a problem known as the *domino effect*[1]. It is the

formation of a set of processes with coordinated recovery points, *a coordinated set* of processes, that is the major goal of both the conversation and the colloquy. The recovery points for a coordinated set of processes is referred to as a *recovery line.* Any transfer of information from a process that is part of a coordinated set to another that is not is termed *smuggling.* Clearly, such communication is not "undone" by backward error recovery, and could disrupt subsequent attempts to provide continued service.

We present here only a summary of the colloquy and an associated construct, the dialog. Complete definitions can be found in Gregory[2]. The *dialog* is a building block for concurrent programs. A dialog is an occurrence in which a set of processes establish individual recovery points, communicate among themselves and with no others, determine whether all should discard their recovery points and proceed (success) or restore their states from their recovery points and proceed (failure), and follow this determination. The determination of success or failure is made by local acceptance tests for each process and a global acceptance test for the set. Dialogs may be properly nested so that the set of participants in the inner dialog is a subset of the participants in the outer. The set of participants in a dialog constitutes a coordinated set.

The *colloquy* is a backward-error-recovery primitive and provides a general framework for describing backward error recovery in concurrent programs. A colloquy is a collection of dialogs. At execution time, a dialog is an interaction among processes. Each individual process has its own local goal for participating in a dialog, but the group has a larger global goal; usually providing some part of the service required of the entire system. If, for whatever reason, any of the local goals or the global goal is not achieved, the actions of the particular dialog are undone. In attempting to ensure continued service from the system, each process

may make another attempt at achieving its original local goal, or some modified local goal through entry into a *different* dialog. Each of the former participants of the now defunct dialog may choose to interact with an entirely separate group of processes for its alternate algorithm. The set of dialogs which take place during these efforts on the processes' part is a colloquy.

The colloquy and dialog, like the rendezvous in Ada, do not exist syntactically but are entirely execution-time concepts. However, we have defined syntactic extensions to Ada that allow each to be programmed[2]. These syntactic extensions are the DISCUSS statement for the dialog and the **dialog_sequence**, a variant of the Ada SELECT statement, for the colloquy.

Provision of recovery lines using colloquys avoids the domino effect. Colloquys also avail the programmer of many powerful facilities for management of backward error recovery. It is tempting to think that this solves all the problems that might arise, and that the syntax for the colloquy can be integrated into a programming language, like Ada, with no further concern. When this is attempted, problems that are at least as serious as the domino effect arise. These problems fall into the general categories of *program structure*, *shared data*, and *process manipulation*, and are discussed under these headings in this paper.

It is important to realize that these problems are not specific to the colloquy. They arise because of the fundamental requirements of backward error recovery in concurrent systems, and occur with the conversation also. We use the colloquy merely as an example. In general, other researchers have either assumed these problems would not occur, or based their research on programming languages with very limited (and impractical) facilities, such as CSP[12]. The problems we discuss here do not arise in these languages because of the relatively simple semantics that such languages have.

3. PROGRAM STRUCTURE

The first problem in merging backward error recovery into realistic programming languages that we address here is the incompatibility between deliberate establishment of recovery lines and the inter-process communications philosophy that most languages employ.

The problems arise because the establishment of recovery lines must be a coordinated activity, agreed upon explicitly by *all* the participants. Most programming languages, however, try to avoid this level of explicit knowledge in communication structures. In Ada, for example, naming is one way only, and, in principle, an Ada task defining an entry does not know which other task is calling the entry during a particular rendezvous. This incompatibility in requirements leads to a general difficulty in program structure. We will discuss here only a particular example, that of providing *service processes*.

A service process provides a service for other processes. It is like a procedure that can only be called in a mutually exclusive manner. With one-way naming, the service process does not know for whom it is providing the service at any time.

The use of service processes is the preferred method in many programming languages for sharing either simple variables or abstractions among multiple processes. The facilities of a service process are used when other processes need some operation performed on the abstraction. The service process makes non-deterministic choices among users awaiting its various services. The services are performed during communication with the individual users, for example during rendezvous in Ada.

Consider now a programming language that makes provision for both service processes and backward error recovery. A process, the *client*, that requires the use

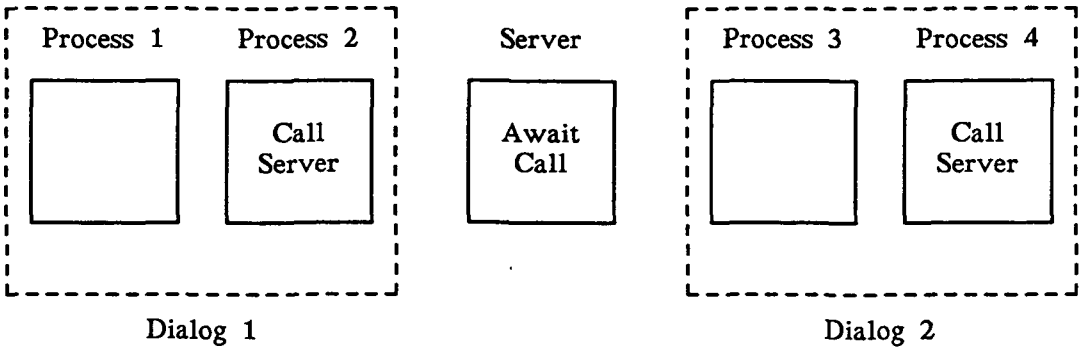


Figure 1 - Inaccessibility Of A Server Process

of a service process, the *server*, must communicate with the server. However, this communication *must* take place within a dialog and this requires an explicit action by the server. The server must enter a dialog in order to provide a service to the client, yet it cannot know which dialog to enter, or when, because it has no knowledge of the state of those processes that might be its clients. It doesn't even know their names. An example is shown in figure 1 in which a server process, outside of all dialogs is idle, yet two client processes are unable to obtain service since each is in a separate dialog.

A solution that might be considered is to arrange for service processes to be active, searching for clients, rather than passive, waiting for clients. A service process might be expected to enter a dialog merely to check to see whether its services are needed. Upon entry, the service process will have to establish a recovery point. Unfortunately, if its services are not needed, it has to await the completion of the dialog before it can continue since it has joined a coordinated set and must discard its recovery point with the rest of the set. This imposes an arbitrary delay on the service process. If its services are required it can provide them but then it still has to await completion of the entire dialog before it can

leave and look for other clients. If dialogs are nested, the service process will have to enter them all but await the completion of each before it can exit. In effect, by actively looking for clients, the service process will be "trapped" in every dialog that it enters until the other processes in the dialog determine that the dialog is complete. We refer to this phenomenon of entrapment until the dialog completes as the *capture* effect. An example is shown in figure 2. The service task is nested within two dialogs but has not further clients. It cannot leave however until both dialogs complete.

We note also the possibility of deadlock with service processes. Two processes that already have provided their services and are ready to continue might be trapped in dialogs. If each dialog requires the attention of the other service process in order to complete, the system will be deadlocked.

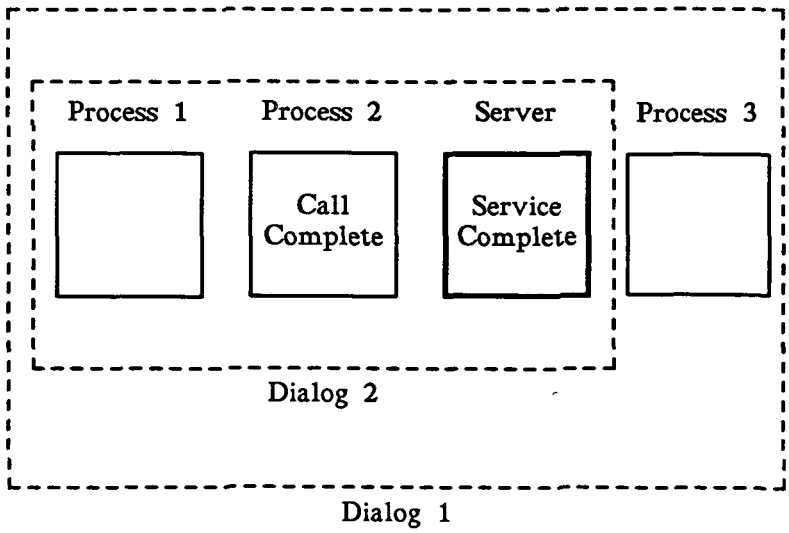


Figure 2 - Service Process Entrapment

An initial approach to this problem that comes to mind would be to require program structures that do not use service processes. This is a severe, perhaps unrealistic restriction given current design techniques for concurrent programs. A second approach might be to designate service processes as somehow special cases, and allow them to operate outside of the dialog rules and arrange their own recovery. This "solution" is easily abused and substantially reduces the compile-time checking of the use of backward error recovery that is possible with the general dialog scheme. We have found *no* satisfactory solution to the problem of the capture effect, and emphasize that it is but one example of the general issue of program structure for systems incorporating backward error recovery.

4. SHARED DATA

A principal means of implicit communication between processes in most programming languages is the use of shared objects. A shared object is any changeable, non-process, non-procedural object that is visible or accessible to more than one process. A shared object that has only one access path or name is called a *shared variable*. In sequential programs, these are called global- or non-local-variables. Shared objects can have multiple names or access paths, either intentionally through *pointers*, or unintentionally through *aliasing*. Ada provides single-access-path and both forms of multi-access-path shared objects. We will not consider the issues that arise with pointers or aliasing here because of space limitations. They are dealt with in depth by Gregory[13].

The problem that shared objects introduce is the possibility of *smuggling*. Since access to shared objects normally is not restricted, it is possible for a process within a coordinated set to access a shared object, and another process, not in the same coordinated set, to access it also. This is uncontrolled communication, and if, as one

would expect, either makes use of the information so acquired, this might defeat backward error recovery.

A solution that suggests itself immediately is to disallow shared objects. This would be unrealistic, however, since programmers *are* going to want exceptions to such a rule. The fact that Ada permits shared objects at all is an indication that they are still considered vital by the community. We do not contend that sharing is essential, only that it seems to be in such demand that we need to deal with it rather than simply define it away. We note, however, that other research areas, such as verification, have also found shared objects very difficult to deal with and prefer to omit them[14].

Besides habit, a practical reason for sharing that is often cited is the need for fast access to shared objects. In addition to the delays that can arise through complex message protocols, there is the potential of incurring further delays in copying parameters, particularly large tables and arrays, in calls to whatever interface is provided for the shared objects.

A final justification for shared objects is the global acceptance test in the colloquy. Without shared objects the global acceptance test would have very little to check.

Given that shared objects must be provided, various approaches can be considered. Unfortunately, several appealing approaches have subtle difficulties associated with them. We will examine these approaches and illustrate their difficulties. Finally, we present a solution that we believe to be workable.

Rather than omitting shared objects altogether, an approach that might be considered is *hiding* of the objects within some process designed to guard them.

Indeed, the designers of Ada advocate this and, in the absence of backward error recovery, it would be the preferred programming paradigm despite the efficiency issue mentioned above. Any other process that needs to access a shared object would be required to communicate with the guardian process (using a rendezvous in Ada for example) and the guardian process would then access the object. This seems quite satisfactory until it is realized that any communication with the guardian has to take place within a dialog. How does the guardian know which dialog it is to enter (if any) at any given time in order to make its services available? A process cannot even make its needs known to the guardian without them both being in a dialog. For a process merely to make its needs known is communication. In fact, it is impossible to write the guardian because there is no way that the guardian can know which process needs it, and in fact, the guardian is just a *service process* with all of the difficulties described in section 3.

Since the use of shared data constitutes communication, and dialogs are designed to provide a recoverable envelope around communication, another approach might be to allow shared objects to be accessed freely but to require that all accesses be within dialogs. Clearly this is totally unworkable since it does not prevent or even detect smuggling.

An approach that provides controlled access to shared objects within a dialog is to require that they be designated explicitly in an *import list*. As well as an import list, this would require that shared objects be accessed only inside a dialog, and that any given shared object only be available to a single dialog at a time. This approach guarantees no conflicting access between parallel dialogs, and guarantees recoverability. The problems with the approach are as follows. It imposes a major limitation on parallelism since only one dialog that uses a given shared object can be permitted to proceed at any given time. All others will have to be suspended. In

addition, it opens up a new opportunity for deadlock since dialogs would be competing in an uncontrolled way for exclusive use of a shared "resource". Further, such a scheme would be extremely inefficient to implement. Each shared object would have to have a locking mechanism associated with it and, since dialogs can be nested, it would be necessary to maintain at least a stack of locks for each object. Attempting to eliminate the stack requirement by not allowing dialogs to be nested defeats a major aspect of backward error recovery, namely *nested recovery regions*. Nested recovery regions are important since an outer region may allow a fault to be tolerated that was *not* tolerated by an inner region. The complexity of the locking mechanism is even worse if distinct read/write locks are considered in an effort to improve the degree of parallelism.

A final problem with the import list approach occurs when a process and a dialog are both nested inside a second dialog. The process and the nested dialog both "think", correctly, that they may access shared objects named in the import list of the outer dialog. Clearly, if they do, smuggling has occurred once again.

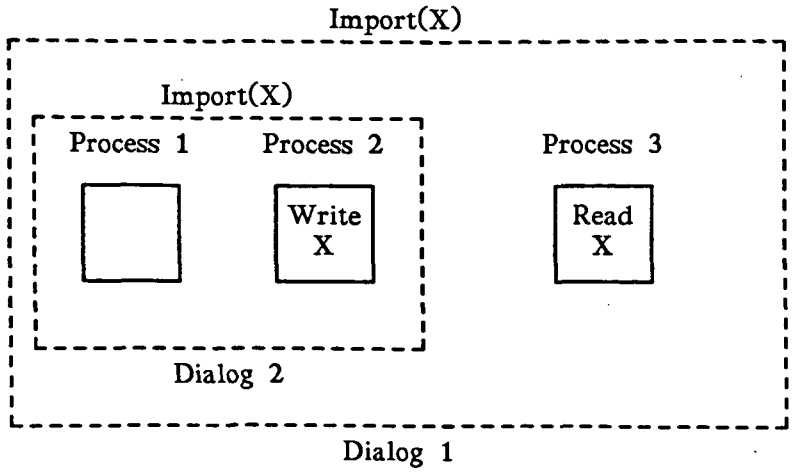


Figure 3 - A Problem With Import Lists

An example is shown in figure 3. Process 3 is reading X while process 2 is writing X, thus information is being smuggled out of dialog 2. The import lists are correct but do not prevent this.

Adequate control of access to shared objects is surprisingly difficult. We introduce now an approach called *image variables* that has none of the difficulties of the *non*-solutions discussed above.

Briefly, the rules of the image variable approach are as follows. Unless explicitly declared to be a shared object, any object is visible only to the process that declared it. However, an object may be declared as shared and then can be accessed by the declarer and other processes but only within a dialog. An object declared as shared presents an *image* within each dialog. Different dialogs reference and manipulate their images of the object *independently*. An object's images are created at the beginnings and merged at the endings of dialogs' existences. As a dialog begins, a new image of the shared object is produced that is conceptually a copy of the image existing in the surrounding dialog, or the surrounding environment if there is no surrounding dialog. If a dialog ends successfully, any change in its image of the object since its inception overrides the surrounding dialog's image of the object. The conceptual copy and overwrite operations are indivisible. If the dialog does not end successfully, the image is discarded.

An image is shared among members of the coordinated set of processes within a dialog but not *among* coordinated sets. Thus, there can be no smuggling. The overhead associated with implementing images occurs only at the entry to and exit from dialogs. These are places where overhead at execution time is to be expected, and the frequency of entry and exit will be substantially less than the frequency of references to shared objects. Thus the overhead will be considerably lower than that which occurs with any of the schemes outlined above. For example, changes to a

table are seen only by those responsible for the changes until they are ready to commit to them. Meanwhile, users of the table carry on with their own image of the table as an unchanging base to work from in performing their own computation. If a dialog fails, its images of shared objects disappear and the images (representing the parallel progress) of other dialogs, including the surrounding dialog, are unaffected. If more than one dialog manipulates a shared object at the same time, and more than one succeed, the value finally taken by the shared object will be that of the image variable of the dialog that succeeded last. Values written earlier will be overwritten. The programmer must take account of this, but the problem is no different from parallel updates to shared objects in programming languages that do not provide backward error recovery, and so it is a familiar problem. It is important to realize that the goal of image variables is to facilitate backward error recovery by preventing smuggling in programming languages that provide shared objects. They are not designed to prevent faults that might occur through other abuses of shared objects. Backward error recovery is an approach to fault tolerance, not an approach to fault avoidance.

5. PROCESS MANIPULATION

Dynamic process creation and destruction, and any facility that allows processes to examine other processes' execution states, are very important programming facilities. It is essential, therefore, that process manipulations be amenable to backward error recovery. All three of these facilities must be considered but, because of space limitations, we consider only process creation here.

The key problem that arises in process creation is the exact disposition of the created process. The creator may require backward error recovery after the creation and so, for the first part of its existence, the created process must be constrained so

that backward error recovery of the creator is possible. However, after the creator has committed to the existence of the created process, the created process must be able to to operate independently of the creator since, in many cases, it will be intended that the created process survive the dialog in which it is created. In fact, the created process might be long-lived, and might even survive the creator. Note that these changes in the status of the *created* process, from severely constrained to independent, are determined by events in the execution history of the *creating* process.

The difficulties in process creation are not limited to the creating and created processes. Process creation is a form of *implicit* communication and is often accompanied by explicit communication with the created process. Knowledge that a process has been created constitutes information about the existence and progress of that process *and* information about the progress of the process responsible for the operation.

If a newly created process has to be "uncreated" as part of backward error recovery, any existence and progress information that might have been communicated to other processes is invalidated. This means, for example, that processes aware of the existence of a created process must be in a coordinated set, or smuggling will have taken place. Smuggling due to process manipulation has to be prevented just as it has to be in other situations.

Thus the problem with process creation is that it must take place within a dialog to allow backward error recovery of the creator, but the created process must be free to proceed independently after the creator commits. The creation of processes through elaboration of declarations causes few problems, and we will not discuss them here. The major difficulties lie in the dynamic creation of processes, such as the execution of an allocator in Ada.

At first sight, it might seem that this problem is trivial. We could require the creation operation to take place inside a dialog, and assume the created process will be suspended until the creator successfully exits that dialog. This has the advantage of preventing the newly created process from engaging in communication with non-members of its creator's coordinated set. The approach also has several drawbacks. First, many languages (and Ada is one) do not have a parameterized process-creation operation. This means that the creator will often want to communicate with the created process explicitly before committing. This cannot be programmed if the new process is suspended. Second, suspending the new process restricts the available parallelism too severely, and could lead to deadlock. For example, other rules of the programming language, such as the task dependency rules in Ada, may prevent the creator's exit from a dialog until the new process has completed execution, which it cannot do if suspended at its beginning.

An alternative approach is to not suspend the newly created process. Analysis of this situation[13] leads to the conclusions that the new process must come into existence within some dialog, and that this dialog must be the one its creator is in while performing the create operation.

Unfortunately, this approach is incomplete. First, as a full member of the creator's dialog, the created process must specify an acceptance test of its own that will be evaluated when it leaves the dialog. However, since it was created in the dialog, it did not enter it in the normal manner (i.e. via a DISCUSS statement). Thus its source text will not contain the necessary acceptance test. Second, it may have been created inside a set of dialogs that are nested to an arbitrary depth. Since the created process cannot tell what this depth is, it has no means of determining how many acceptance tests it needs or what they should be.

A refinement of this approach that we suggest is as follows. A created process is divided into two parts, the *preliminary* and *main* parts. The preliminary part is executed after creation and before the creator exits the dialog in which the creation took place. The boundary between the two parts is marked by an acceptance test. Evaluation of this acceptance test by the created process occurs as the dialog ends and is its primary mechanism for vetoing its own creation. Once this dialog is complete, execution of the main part begins.

Recall that the created process could have been created within a set of dialogs nested to an arbitrary depth. In the approach that we propose, the created process' viewpoint during execution of the main part is that these outer dialogs do not exist. From the viewpoint of other processes, it continues to be nested within the "current" dialog of its creator until it is no longer possible for its creation to be undone by backward error recovery. Within both of its parts, the created process may communicate with any members of its creator's "current" coordinated set, but must enter a new dialog to do so, along with those with which it wishes to communicate. As the creator exits dialogs, so the set of processes with which the created process can communicate changes.

Naturally, failure of any of the dialogs in which the creator is a participant causes the created process to be "uncreated". Since the set of processes with which it could communicate is delimited by the coordinated set(s) of its creator, this backward error recovery is *always* possible if required. Smuggling of information about the existence of the created process *cannot* occur.

The remaining element of the approach we propose is a function called FINAL. Since the created process is unaware of its involvement in various dialogs, it has no local acceptance test(s) that permit it to check its operation as it exits these dialogs. The purpose of the FINAL function is to provide a single local acceptance test for

the created process that is used upon exit from the outermost dialog. The FINAL function is specified along with the specification of the created process, and executed asynchronously for the created process on exit from the outermost dialog. For the outermost dialog to succeed, the local acceptance tests of all of its participants, the global acceptance test, and the FINAL functions for all created processes must evaluate to **true**.

6. CONCLUSION

This paper has addressed some of the general issues that arise when an attempt is made to integrate backward error recovery into a realistic programming language.

A major inconsistency exists between the preferred structure of concurrent programs and the structure necessary for backward error recovery using planned establishment of recovery lines. This inconsistency is illustrated by the capture effect for service processes. We have found no satisfactory resolution for this problem.

Shared data is a significant source of smuggling in concurrent systems. Even limiting consideration to just shared variables, we showed that there are several approaches that appear to solve this problem, but each of them is inadequate, and has to be rejected. We introduced the concept of image variables as a compromise and showed that image variables are a workable solution to the shared variable part of the shared data problem.

Finally, the problem of smuggling arises with process manipulations also. We concentrated in this paper on the manipulation known as dynamic process creation, and showed how smuggling can occur in this context. Manipulations in which

processes' execution states are examined and those in which processes are destroyed were not considered here but are extremely important nevertheless. Once again, there are several non-solutions that one might consider in seeking a solution. We have presented a workable solution to this part of the process manipulation problem.

Our overall conclusion is that the concepts of backward error recovery cannot be merged *naively* into existing realistic programming languages. All of the problems described here derive from the rapid growth in size and complexity of programming languages with no attention to backward error recovery. The final solution might be the design of an entirely new programming language with backward error recovery as its starting point.

REFERENCES

(1) Randell B., "System Structure for Software Fault Tolerance," *IEEE Transactions on Software Engineering*, **SE-1**(2), pp. 220-232 (June 1975).

(2) Gregory S.T., Knight J.C., "A New Linguistic Approach to Backward Error Recovery," *Digest of Papers FTCS-15: Fifteenth Annual Symposium on Fault-Tolerant Computing*, pp. 404-409 (June 1985).

(3) *Reference Manual for the Ada Programming Language*, ANSI/MIL-STD-1815A (22 January 1983).

(4) Anderson T., Barrett P.A., Halliwell D.N., Moulding M.R., "An Evaluation of Software Fault Tolerance in a Practical System," *Digest of Papers FTCS-15: Fifteenth Annual Symposium on Fault-Tolerant Computing*, pp. 140-145 (June 1985).

(5) Horning J.J., et al., "A Program Structure for Error Detection and Recovery," pp. 171-187 in *Lecture Notes in Computer Science 16*, (ed. E. Gelenbe and C. Kaiser), Springer-Verlag, Berlin (1974).

(6) Anderson T., Kerr R., "Recovery Blocks in Action: A System Supporting High Reliability," *Proceedings of 2nd International Conference on Software Engineering*, San Francisco (CA), pp.447-457 (October 1976).

(7) Shrivastava S.K., "Sequential Pascal with Recovery Blocks," *Software - Practice and Experience* **8**(2), pp. 177-185 (March 1978).

(8) Shrivastava S.K., Akinpelu A.A., "Fault Tolerant Sequential Programming Using Recovery Blocks," *Digest of Papers FTCS-8: Eighth Annual Symposium on Fault-Tolerant Computing*, Toulouse, p. 207 (June 1978).

(9) Lee P.A., "A Reconsideration of the Recovery Block Scheme," *Computer Journal* **21**(4), pp. 306-310 (November 1978).

(10) Hecht H., "Fault-Tolerant Software," *IEEE Transactions on Reliability* **R-28**(3), pp. 227-232 (August 1979).

(11) Lee P.A., Ghani N., Heron K., "A Recovery Cache for the PDP-11," *IEEE Transactions on Computers*, **C-29**(6), pp. 546-549 (June 1980).

(12) Hoare C.A.R., "Communicating Sequential Processes," *Communications of the ACM* **21**(8), pp. 666-677 (August 1978).

(13) Gregory S.T., *Programming Language Facilities for Backward Error Recovery in Real-Time Systems*, Ph.D. Dissertation, Department of Computer Science, University of Virginia, (1986).

(14) Amber A.L., Good D.I., Browne J.C., Burger W.F., Cohen R.M., Hoch C.G., Wells R.E., "Gypsy: A Language for Specification and Implementation of Verifiable Programs," *SIGPLAN Notices* **12**(3), pp. 1-10 (March 1977).

Appendix 4

On The Implementation And Use Of Ada On Fault-Tolerant Distributed Systems

To Appear In

IEEE Transactions on Software Engineering

ON THE IMPLEMENTATION AND USE OF Ada[1] ON FAULT-TOLERANT DISTRIBUTED SYSTEMS

John C. Knight and John I. A. Urquhart

Affiliation Of Authors

Department of Computer Science
University of Virginia
Charlottesville
Virginia, 22901

Footnote

1. Ada is a trademark of the U.S Department of Defense.

Financial Acknowledgement

This work was supported by the National Aeronautics and Space Administration under grant number NAG-1-260.

Index Terms

Distributed systems, fault tolerance, Ada, highly-reliable systems, tolerance of processor failure.

Address For Correspondence

John C. Knight
Department of Computer Science
University of Virginia
Charlottesville
Virginia, 22901

Abstract

In this paper we discuss the use of Ada on distributed systems in which failure of processors has to be tolerated. We assume that tasks are the primary object of distribution, and that communication between tasks on separate processors will take place using the facilities of the Ada language. It would be possible to build a separate set of facilities for communication between processors, and to treat the software on each machine as a separate program. This is unnecessary and undesirable. In addition, the Ada language reference manual states specifically that a system consisting of communicating processors with private memories is suitable for executing an Ada program.

We show that there are numerous aspects of the language that make its use on a distributed system very difficult if processor failures have to be tolerated. The issues are not raised from efforts to implement the language, but from the complete lack of semantics in Ada defining the state of a program when a processor is lost.

We define appropriate semantic enhancements to Ada, and describe the extensive modifications to the execution support required for Ada to implement these semantics. A program structure making use of these semantics is defined that includes all the necessary facilities for programs written in Ada to withstand arbitrary processor failure. If the required program structure is used and the necessary support system facilities are available, continued processing can be provided.

I INTRODUCTION

Ada [1] was designed for the programming of embedded systems, and has many characteristics designed to promote the development of reliable software. In this paper we examine the problem of programming distributed systems in Ada. In particular, we are concerned with the issues that arise when some form of acceptable processing must be provided using the facilities remaining after a hardware failure. This is important in *crucial* applications; applications such as defense or avionics systems where total failure of the computer system could have a very high cost.

By a distributed system we mean a set of processors linked together by a high-performance communications bus. Each processor would have its own memory, and devices such as displays, sensors, and actuators would be connected to the bus via dedicated microprocessors; these devices would be accessible from each processor.

Although much research has been undertaken in recent years to produce reliable computers [2, 3], these machines may still fail. Moreover, lightning, fire or other physical damage can cause a processor to fail no matter how carefully it is built. One of the advantages of distributed processing is that a hardware failure need not remove all the computing facilities. If one processor fails, it is possible, at least in principle, for the others to continue to provide service.

In this paper, we assume that communication between processors on a distributed system will be implemented using a protocol that conforms to the ISO standard seven-layer Reference Model [4]. The kind of hardware failure that we are concerned with, the total loss of a processor with no warning, is not addressed by these protocols and must be dealt with separately.

We assume that the processors have the *fail stop* property [5]. This means a processor fails by stopping and remaining stopped. All data in the local memory of the processor will be assumed lost. Thus the case of a processor failing by continuing to process instructions in an incorrect manner and providing possibly incorrect data to other processors will not be considered. Given this assumption, error detection reduces to detecting that a processor has stopped. Error recovery is simplified by the knowledge that although the data in the failed processor's memory is lost, data on the remaining processors is correct.

A distributed system that is to be highly reliable will be built with a redundant bus structure. A complete break in the bus system that isolates some subset of the processors

(i.e. the network becomes partitioned) is very serious though extremely unlikely given multiple routes and replication. The issues that arise in this case are different from those arising from processor failure. They are beyond the scope of this paper and will not be dealt with here.

We begin in section II by looking at two approaches that might be taken to tolerating the loss of a processor and providing continuation. One approach relies on the support system providing a virtual target in which failures do not take place. In this approach, the application program is unaware of any failures that occur. In the second approach, the support system provides minimal facilities and the application program is expected to cope with the situation. The second approach is the subject of the remainder of the paper. In section III we consider the facilities that are required for continuation and discuss their relative absence in Ada. We show that the problem centers around the lack of semantics describing distribution and failure for an Ada program. Appropriate distribution and failure semantics for Ada based on the requirement that the language syntax remain unchanged are presented in section IV. An execution-time system to provide these semantics is presented in section V. We show how these semantics can be used in Ada programs in section VI and discuss their implementation and execution-time performance in section VII. Our conclusions are presented in section VIII.

II APPROACHES TO FAULT TOLERANCE

At the interface provided to the application program, there are two different approaches that can be taken when attempting to provide tolerance to hardware failures. In the first approach, the loss of a processor is dealt with totally by the software providing the interface. Any services that were lost are assumed by remaining processors, and all data is preserved by ensuring that multiple copies always exist in the memories of the various machines. We refer to this approach as *transparent* since, in principle, the programmer is unaware of its existence. This is the approach to building fault-tolerant distributed Ada systems being pursued by Honeywell [6, 7].

Transparent continuation has several advantages. For example, the programmer need not be concerned with reconfiguration and need not know about the distribution. It follows that the same program can be run without modification on different systems with different distributions.

A disadvantage of transparent continuation is that the programmer cannot specify degraded or *safe* [8] service to be used following processor failure. Since such service cannot be specified by the system, transparent continuation must always provide full service or no service. In many crucial systems this is not acceptable. In practice, this approach may not even be permitted by agencies regulating crucial systems. Such agencies frequently require that the precise actions that will be taken following failure must be stated explicitly, and that degraded service must always be provided if any computing facilities exist after a failure.

A second disadvantage of transparent continuation is its overhead. Since failures can occur at arbitrary times, the support software must always be ready to reconfigure. Duplicate code must exist on all machines and up-to-date copies of data must always be available on all machines. The overhead involved in this process is considerable.

This overhead will also be influenced by the program that is to be distributed. An unfortunate consequence of transparent continuation is that distribution and fault tolerance are not taken into account at the top level of design. Thus the program may be written in a way that makes distribution awkward and greatly increases the overhead.

In the second approach to dealing with the loss of a processor, only minimal facilities are contained in the interface provided to the application program. The fact that equipment has been lost is made known to the program which is expected to deal with the situation. We will refer to this approach as programmer-controlled or *non-transparent* since the programmer is responsible for providing the fault tolerance.

Non-transparent continuation has several disadvantages. For example, the programmer must be concerned with reconfiguration, and must either specify the distribution or be prepared to deal with any distribution provided by the system. Also, the program will depend on the system; at least the reconfiguration parts will.

The disadvantages are out-weighed by the fact that the service provided following failure need not be identical to the service provided before failure. The programmer has complete control over the services provided by the software and the actions taken by the software following failure. Alternate, degraded, or merely safe service can be offered as circumstances dictate.

In the remainder of this paper we will consider only the non-transparent approach. We assume that the actions to be taken by each processor following a failure are specified

within the software executing on that processor.

III CONTINUATION REQUIREMENTS AND Ada

In the non-transparent approach, the programmer must deal with both distribution and continuation. Continued service after one or more processor failures requires that the following actions be performed:

(1) *Detect Failure*
Processor failure must be detected and communicated to the software on each of the remaining processors.

(2) *Assess Damage*
It must be known what processes were running on the failed processor, what processes and processors remain, and what state the processes that remain are in.

(3) *Select A Response*
Information must be provided so that a sensible choice of a response can be made. This choice will normally depend on which processors and processes remain, but in many applications the choice will also depend on other variables and their values would have to be known.

(4) *Effect The Response*
Once a response has been decided on, it must be possible to carry it out.

We note that these actions depend strongly on data that is consistent across all machines.

In this section we discuss, in detail, distribution, the actions listed above, and the provision of consistent data. It will be seen that many of the required continuation facilities are absent from Ada.

Distribution

It is essential that software to be used following the failure of a particular processor not be resident in the memory of that processor (otherwise the system would be centralized). To achieve this separation, the programmer *must* control the placement of both

the primary and replacement software.

It is not sufficient to be able to specify that the primary and replacement software be on different machines. It is essential that the programmer be able to designate exactly which machine each will reside on. If this is not possible, the reconfiguration software will have to be prepared to deal with the loss of any combination of the primary tasks when a machine fails.

It is not sufficient merely to be able to control the allocation of software to processors. There must be explicit semantics for distribution and these semantics *must* deal with processor failures. For example, in Ada if there are multiple tasks of a particular task type and they are executing on different processors, a separate copy of the code must be required for each processor. Otherwise, an implementation could provide a single copy of the code that was shared by all processors; for example by fetching a copy when a new task body is elaborated. This would be satisfactory if there were no failures. However, failure of the processor containing the original copy of the code would then suspend all subsequent elaborations.

The choice of objects to be distributed is an important question in the design of a distributed system. Thus a programming language that is to provide continuation must specify what units can be distributed, and exactly how the distribution is to be done. We refer to these as the *distribution semantics*. In addition, the language should include a syntax for expressing distribution.

Ada has a tasking mechanism and, according to the Ada Reference Manual [1], it is intended that tasks be distributed in an Ada program:

> Parallel tasks (parallel logical processors) may be implemented on multicomputers, multiprocessors, or with interleaved execution on a single physical processor.

Also, it is clear from the requirements for the language [9] and from the Ada Reference Manual [1] that the tasking facilities are intended to be used for all task communication and synchronization even when different physical processors are executing the tasks involved.

It would be possible to consider distributing complete programs [10]. This would require the development of a separate mechanism for inter-task activities between computers using some form of input and output. However, this would be equivalent to communication by shared memory and would be less secure than the existing language

facilities because compile-time checking would not be possible. Further, it would be specific to the program for which it was written, and a program design change that required a task to be relocated to a different computer could force substantial changes in those tasks that communicate with it. An excellent discussion of the disadvantages of distributing complete programs is given by Cornhill [7].

No facilities are defined in Ada to control the distribution of tasks. Surprisingly, although there are representation clauses to control the bit-level layout of records, to allow the placement of objects at particular addresses within a memory, and to associate interrupt handlers with specific machine addresses, there is no explicit mechanism for control of distribution in Ada. Although Ada was designed for distributed systems, neither the syntax nor the semantics for task distribution is defined.

Failure Detection

Normally, a facility for failure detection would be provided by the underlying system. The application software will have to be informed of the failure so that it can take the necessary actions to continue. The programming language should provide an interface that would allow the failure to be signaled. No specific interface is provided by Ada to allow software to be informed of processor failure.

It is possible to inform the software by raising an exception or generating an interrupt; the latter using an entry call as its interface. In either case, appropriate placement of the corresponding exception handler or **accept** statement is a problem since it must be assured that they will be executed promptly. Also, the necessary exception and entry names are not predefined and so their use is neither standardized nor required.

Damage Assessment

Clearly, the damage sustained as a result of a processor failure includes loss of the services that were provided by the software that was executing on the processor that failed. It also includes loss of the data contained in the memory of the failed processor. In addition, the failure of a processor can affect the software that remains. The programming language definition must specify exactly which part of the remaining software will be affected and exactly how it is affected. We term this the *failure semantics* of the language.

Ada does not define any failure semantics. However, it is clear from the rules of the language that a great deal of the remaining software can be affected by the loss of a processor. Broadly speaking, this can happen is two ways. A task can be suspended during communication waiting for a message that will never arrive, and a task can lose part of its context.

The problems that arise in task communication, the first way that a task can be affected by processor failure, are best illustrated by an example. Consider an Ada program that contains two tasks, A and B, executing on different processors. Suppose that task A has made a call to an entry in task B, and that B has started the corresponding rendezvous. If B's processor now fails, task A will remain suspended forever because the rendezvous will never end. Since the failure takes place after the start of the rendezvous, a timed or conditional entry call will not avoid the difficulty. In addition, task A cannot distinguish this situation from a slowdown in task B caused by a temporary increase in activity on its processor.

Similar problems arise throughout Ada, both in explicit communication such as the rendezvous, in implicit communication such as task activation and termination, and even in referencing shared variables. A detailed examination of these situations is given in [11].

The second way that a task may be affected by processor failure is the loss of part of its context. In block structured languages, a program unit can assume the existence of an instance of all objects in the surrounding lexical blocks. When a system is distributed, it is possible to have a given program unit on one processor and the objects defined by one of its surrounding lexical blocks on another processor. A task in Ada relies on several contexts; the context of the body, the creator, and the masters [1]. All of these contexts may be different, and each of them may be lost due to processor failure. Ada defines no semantics for these situations.

In summary, the damage following processor failure will include lost services and lost data. The remaining software may be affected by the failure and the effects include the permanent suspension of tasks on remaining processors for a variety of reasons, and the loss of contexts of some tasks. These effects could be quite extensive. As presently defined, Ada provides no definition of what software will be affected or how it will be affected; Ada has no failure semantics.

Selecting a Response

The selection of a response is made by the application program. In order to avoid a centralized system we suggest that each processor should contain a unit that will select the response for the processes on that processor. Clearly the information on which the response is based must be consistent across processors. Usually, the response will depend on the processes lost by the failure, so that the unit which selects the response should be able to obtain that information.

In Ada this could be done in several ways. In the example described below, the unit which selects the response on each processor is a task, which waits at an accept until a failure occurs. After a failure, the system calls the corresponding entry and passes the information about the loss of processes as parameters to the entry call.

Effecting a Response

The purpose of effecting a response is to provide replacement services for the services that were lost. The source of the new services will have to be software that resides on machines that remain after the failure. For any programming language, the failure semantics must *guarantee* the existence of this remaining software. It there are no failure semantics or the failure semantics do not require adequate software to survive a failure, then it may not be possible to effect any response. To be useful, the states specified by the failure semantics for processes that remain but are damaged must be such that they allow some form of recovery. At the very least, it should be possible to remove such processes.

Although Ada has no failure semantics, for the purposes of discussion, we will assume that it does, and examine the subsequent issues that arise in effecting a response. Some facilities of the language can be used to effect a response while others limit what can be done.

In general it will be necessary for the software effecting the response to be able to communicate with the existing software. Clearly the software effecting the response cannot call an entry in an application task and wait for the task to reach the corresponding accept; it might never reach it. The alternative is for the application task to check with the software that effects the response whenever it reaches a state that could be affected by

failure. A better solution would be for the software effecting the response to be able to raise an exception in the application task. Although this is not possible in Ada, the effect of communicating with a task so that it can modify its behavior because of a failure can be obtained, at some additional cost, by aborting the original task and starting a replacement task which incorporates the modifications. The basic problem in Ada is that there is no way of getting the "attention" of a task unless the task cooperates.

Once the problem of communicating with a process has been overcome it is still necessary to program the desired modifications. Often the communication paths used before failure will have to be redirected. The rendezvous in Ada is asymmetric; a calling task needs to know the name of any task containing an entry it wishes to call, but a called task need not know the names of tasks that will call it. If a calling task has to be replaced because of a failure, the replacement can call the same entry that was called by the lost task. The entry is still available in the same task that was being called before the failure. Thus redirection is trivial if a calling task is lost.

However, if a called task has to be replaced because of a failure, the replacement cannot be given the same name as the task that was lost. This would duplicate the definition of a task name in the same scope. Thus, in this case, redirection is more involved. The replacement called task will have to have a different name and, more importantly, all the calling tasks (that may not have been replaced) will have to begin using a different name in their entry calls.

This difficulty is not quite as serious for tasks created by allocators. Since assignment is allowed for access variables, communication can be redirected by assigning a value representing a replacement task to an access variable used to make entry calls. Two problems then arise. First, the entire program has to be written using access variables to reference tasks. Second, a replacement task has to be of the *same* type as the primary task and this makes the provision of degraded service difficult.

Although Ada allows the software effecting the response to create and start replacement tasks, the scope rules of the language require lexical placement of the replacement software within the scope of the software effecting the response. In fact, since the software effecting the response must be able to see all of the tasks on a processor, it may be necessary to spread this software across several tasks.

Consistent Data

When a failure is detected, the reconfiguration software on each processor will assess the damage, and select and effect the response. Each of these activities will normally depend on data surviving the failure and this data must be identical on *all* machines. For example, usually the selection of a response will depend on which processors and processes remain, but in many applications the choice will also depend on other variables and their values will have to be known. The altitude of an aircraft, for example, might determine the actions to be taken when part of the avionics system is lost. If different processors have different values for the altitude after a failure, the responses they select may work at cross-purposes.

Ada provides no facility for maintaining consistent data across machines. Pragma **shared** does require that all copies of a shared variable be updated whenever any copy is updated, but it does not guarantee that if a machine failure occurs during the update process either all or none of the remaining copies will have been updated.

IV DISTRIBUTION AND FAILURE SEMANTICS FOR Ada

It is possible to overcome the difficulties discussed above. The first step is to define distribution and failure semantics for Ada. While it is not difficult to choose a reasonable meaning for distribution, the problem of what to do with damaged tasks is much more difficult. It must be emphasized that the semantics suggested in this section were chosen so as to follow the existing semantics of Ada as closely as possible, and to allow the Ada syntax to remain unchanged.

Distribution Semantics

The primary aim of distribution semantics is to avoid the possibility of a centralized distributed system. It will be assumed that only tasks will be distributed. The distribution of a task T to a processor P will be taken to mean that the task activation record for T and all of the code for T will be resident on P. There is no theoretical reason to limit distribution to tasks, and we agree with Cornhill [7] that other objects could be considered. However, tasks are natural candidates for distribution and were intended for distribution in the requirements for Ada, and, for simplicity, we limit our discussion to tasks.

Failure Semantics

In section III it was pointed out that the failure of a processor may affect tasks running on the remaining processors, and that many language features can cause these problems. The difficulties do not arise because tasks were lost when the processor failed. Any task could be removed from an Ada program at essentially any point *without* processor failure by execution of an **abort** statement. Rather, the difficulties arise because the semantics of the language fail to deal with the situation. Ada semantics *are* precisely defined for tasks being aborted and for the subsequent effects on other tasks, and the execution-time system is required to cope with the situation. We suggest therefore that failure semantics be defined so that the state of the program following processor failure be identical to the state that would have occurred if the tasks that were lost had been aborted. This would allow the language-dependent part of the damage following processor failure to be treated using existing language facilities.

This choice of failure semantics leads to a new problem: the status of the main program following failure. By definition all tasks in an Ada program whose masters are not library packages depend on the main program. If failure of a processor is to be treated as if **abort** statements had been executed on the lost tasks, then a serious problem arises with the main program. When a task is aborted, all its dependents are aborted also. For any task lost through failure this is reasonable. It means that all the dependents that were not lost with the task have to be aborted by the system. However, if the main program is lost, this implies that all the tasks that depend on the main program (that is all except library tasks) will have to be aborted. This could effectively remove the entire program. Clearly this is unsatisfactory. We suggest therefore that the main program has to be treated as a special case. For the main program, and only for the main program, the execution-support system will have to create an exact replacement if the main program is lost through processor failure. To ease the overhead that this involves, we suggest that the sequence of statements in the **begin end** part of the main program be limited to a single **null** statement. The main program should be used only to define global objects and tasks that will manipulate these objects.

V A SUPPORT SYSTEM FOR FAULT-TOLERANCE IN Ada

Although Ada does not support the facilities required for continuation explicitly, fault tolerance can be achieved if the execution-time support structure is suitably modified to detect failure, implement the failure semantics described in section IV, and provide

consistent data. In this section we discuss the necessary modifications and in section VI we show how they are used with Ada.

Failure Detection

Failure detection could be performed by hardware facilities over and above those provided for normal system operation. Alternatively, failure could be detected by system software. The hardware option is less desirable because it requires additions to existing or planned systems, and the detection hardware itself could fail. We suggest therefore the use of software failure detection.

Software failure detection can be either active, relying on regular liveness checks, or passive, relying on some form of timeout. We suggest the use of active software failure detection. In an active system, some kind of inter-processor activity is required periodically and if it ceases, failure is assumed. The messages that are passed are usually referred to as *heartbeats*.

Implementing Failure Semantics

The mechanism that we propose for implementation of the Ada failure semantics defined in section IV together with the heartbeat mechanism, is shown in figure 1. The failure semantics require that the loss of a processor have the same effect as aborting all tasks on that processor. Implementing this requires that the affected tasks be known and that the semantics of abort be applied.

Whenever any communication takes place between tasks on different processors, the execution-time support system on the processor starting the communication records the details in a *message log*. Whenever a failure is detected, each processor checks its message log to see if any of its tasks would be damaged by the failure (permanently suspended for example). If any are found, they are sent fake messages. They are called "fake" because they are constructed to appear to come from the failed processor but clearly do not. The message content is usually equivalent to that which would be received if the lost task had been aborted. In this way, each processor is able to ensure that none of its tasks is permanently damaged, and the action following failure for each remaining task is that which is associated with an abort. It often takes the form of raising an exception.

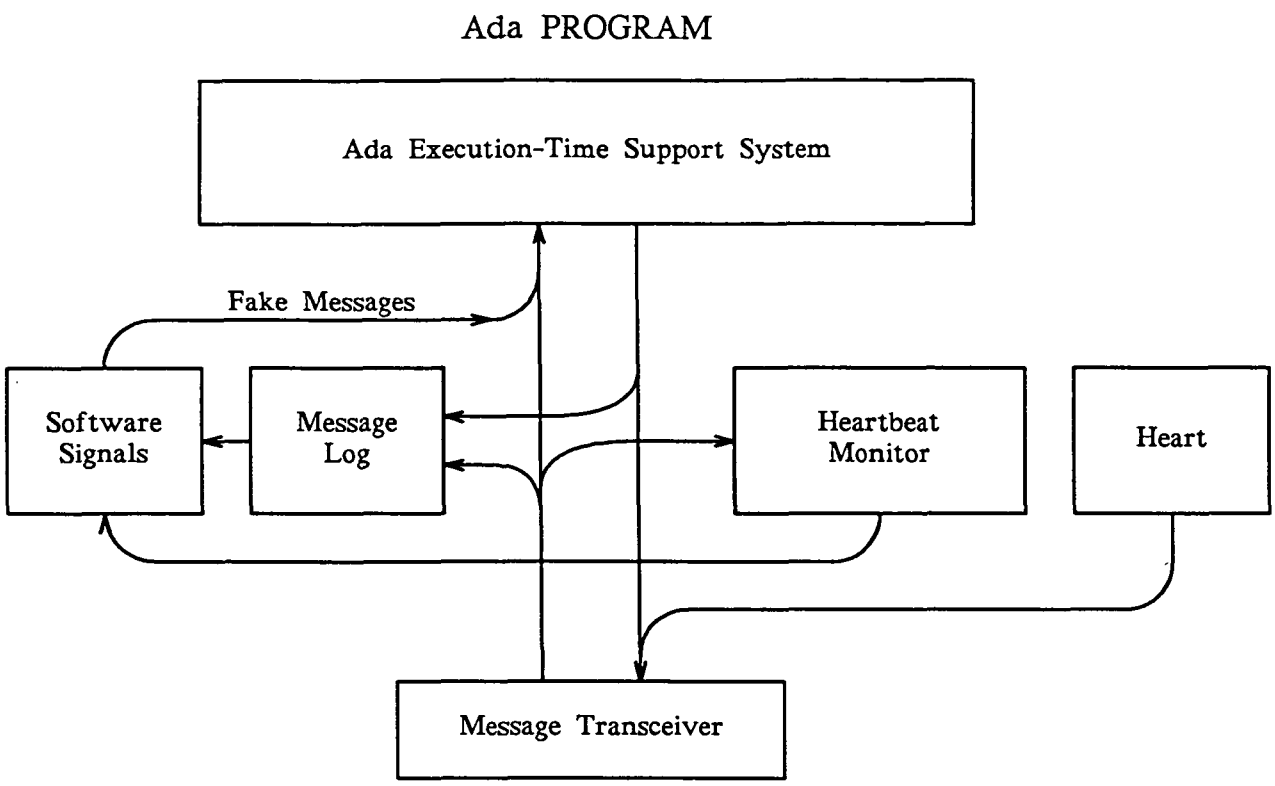


Fig. 1. Implementation Model For Each Processor

Clearly it is possible for unsuspecting tasks to attempt to rendezvous with tasks on the failed processor after failure has been detected, signaled, and other communications terminated. This situation can be dealt with easily if the execution-time support system returns a fake message immediately, for example raising TASKING_ERROR, indicating that the serving task has been aborted and that rendezvous is not possible.

Provision of Consistent Data

Since consistent data across machines is essential to allow a response to be chosen and implemented, the execution-time support system for Ada must provide a mechanism for ensuring that data can be distributed reliably. A *two-phase-commit* protocol [12, 13] can be used, and we propose that an implementation of it be included in the execution-time system along with the message log and heartbeat mechanism.

VI FAULT TOLERANCE IN Ada

In this section we show how a fault-tolerant Ada system can be built using the support mechanisms just described. As was shown in section III, Ada does not provide any specific facilities to support this type of fault tolerance. Existing features of the language that were not designed for the purpose have to be used to interface with the modified support system.

Distribution

Although Ada has no specific facilities for control of task distribution, limited control can be achieved using either an implementation-dependent pragma or an implementation-dependent address clause.

The pragma could have a machine identification as a parameter and be required to appear in the specification of the task or task type to which it applies, as is done with the predefined pragma **priority**. The distribution pragma would require the compiler to generate instructions and loader directives for the designated task that would ensure that it is placed in the required machine. Alternatively, for tasks created by allocators, the distribution pragma could be required to appear in a declarative part; it would apply only to the block or body enclosing the declarative part (similar to the predefined pragma **optimize**). Its effect would be to cause all tasks created by allocators in the block or body to be distributed to the machine designated by the pragma.

Address clauses could be used similarly. For example, in a particular implementation, the identifier parameter in the address clause could be interpreted as a task name, and the expression parameter as a machine designation.

Failure Detection

When failure of a processor is detected by the heartbeat mechanism, this information must be transmitted to the software running on each remaining processor so that reconfiguration can take place. The information is available to the execution-time support software in some internal format, but it has to be transmitted to the Ada software using an existing feature of the language.

The approach to signaling failure that we have chosen is to view the required signal as being like an interrupt, and transmit the information to the Ada software by a call to a predefined entry. This raises the problem of where the entry should be defined to ensure execution. We propose, therefore, that a dedicated task of the highest priority (RECONFIGURE_I) containing an entry with a single parameter be defined on each processor (I is the processor number). The **accept** statement associated with the entry will be in an infinite loop. This task will normally be suspended on the **accept** statement for the entry and, when a failure is detected by the heartbeat mechanism, a call to the entry will be generated. The parameter passed will designate which element of the system's hardware has failed. The task will then be activated and will contain code within the **accept** statement to handle reconfiguration.

A general form of the body of the reconfiguration task is shown in figure 2. Since this task is in an infinite loop, it returns to the **accept** statement once a particular failure has been dealt with. Thus subsequent failures will be dealt with in the same way and, in principle, any number of sequential failures can be dealt with. Further, if physical damage removes more than one processor at the same time, the remaining processors will notice the loss of heartbeats in some order (which must be guaranteed by the heartbeat mechanism to be the *same* order), and calls to the entry in the reconfiguration task will be generated sequentially. Thus multiple failures occurring together will be dealt with as if they had occurred in some sequence.

We note that an exception cannot be used to signal failure. The difficulty is that Ada provides no way to ensure that the task in which the handler is defined will be executing

```
task body RECONFIGURE_I is
begin

   -- Initialization code.
   loop
      accept FAILURE(WHICH : in PROCESSOR) do
         -- Code to handle hardware failures.
      end accept;
   end loop;

end RECONFIGURE_I;
```

Fig. 2. Body Of Task RECONFIGURE_I.

when the exception is raised. What is required is a task that is idle until the failure has to be signaled. This is best achieved by having the task wait at an entry and signaling failure with an entry call.

Damage Assessment

As a consequence of the failure semantics defined in section IV, damage will be limited to lost tasks and the data associated with those tasks. No remaining tasks will be suspended, and all remaining tasks will have their complete contexts. However, the tasks and data that were lost need to be determined. Since the programmer controls distribution this is quite simple.

The information about which tasks are on which processors could be maintained and referenced in three ways: as a table maintained by the execution-support system; as a table maintained by the program itself; or maintained implicitly within the program. In the third case, if all task-to-processor assignments are known at compile time and do not change, the code used for reconfiguration following failure can be written with the distribution information as an assumption. There is no clear advantage to any of these methods. The choice in any particular case is implementation dependent.

Selecting and Effecting The Response

The creation and deletion of tasks that might be required as part of effecting a response is easily achieved in Ada using allocators and the **abort** statement. Thus the particular **accept** statement within the reconfiguration task that is executed for a given failure can create and delete whatever tasks are needed to provide replacement service. A simpler approach to providing replacement software is to arrange for the required replacement task to be present and executing before the failure, but suspended on an entry. Such a task would not consume any processing resources although it would use memory, but it could be started by the reconfiguration software very quickly and easily by calling the entry upon which the replacement task is suspended. A general form for a replacement task is shown in figure 3.

Redirection of communication to replacement software that has been started following a failure has to be programmed ahead of time into tasks that call entries in tasks that might

```
task REPLACEMENT_SERVICE is
   pragma distribute(PROCESSOR_I);
end REPLACEMENT_SERVICE;

task body REPLACEMENT_SERVICE is
begin
   -- Code necessary to initialize this replacement service.
   accept ABNORMAL_START;
      -- This task will be suspended on this entry until it is
      -- called by RECONFIGURE_I following failure. The
      -- code following the accept statement provides the
      -- replacement service. Any data required can be
      -- obtained from DATA_CONTROL_I.
end REPLACEMENT_SERVICE;
```

Fig. 3. General Form Of A Replacement Task.

fail. It will be necessary for the reconfiguration task on each processor to make status information about the system available to all tasks on that processor. Each task must be prepared to deal with a TASKING_ERROR exception after making a call to an entry on a remote machine in case the entry has changed because of failure.

Consistent Data

Algorithms for the selection of a suitable response, and the algorithms used in that response, depend for their correct operation on having appropriate data available. Typically, each piece of data being manipulated by a program for an embedded application can be regarded as either *expendable* or *essential*. For example, partial computations and sensor readings would be expendable whereas navigation information and the status of weapons would be essential.

Expendable data need not be preserved across machine failures. A partial computation, for example, is only of value to the expression generating it. Replacement software that will be used following a failure can recompute any expendable values. A sensor value, for example, is usually useful only for a short time and a suitable replacement value can be obtained by reading the sensor again. We suggest that any data items that the programmer considers expendable be given no special attention, and that the replacement software be written with the assumption that these data items are not available.

Essential data does need to be maintained across machine failures. In an Ada program this could be implemented in two ways. First, data items that the programmer considered to be essential could be marked as such (perhaps by a pragma), and the system would then be required to ensure that copies of these data items were maintained on all machines. Each time the data item was modified, all the copies would be updated. In the event of failure, one of the backup copies could be used immediately. This is simple for the programmer but potentially inefficient. Consider for example a large array that was designated as essential. If it were being updated in a loop, as each element was changed, it would be necessary to update all the copies of that element. The entire overhead associated with maintaining consistent copies would be incurred for each element change. In practice in order to allow reconfiguration, it would probably be adequate to wait until all the elements of the array had been modified and then update all the copies of the array at once.

A second approach is to provide the programmer with the tools to generate consistent copies across machines. In this way, not only the data items to be preserved but also the times during execution when copies will be made will be under the programmer's control. We suggest that this could be done by providing a data management task, DATA_CONTROL_I, on each processor. The data management tasks together would maintain a consistent set of essential data across all processors using a two-phase-commit protocol. When a task had produced some essential data, it would pass that data to the local data management task by calling an entry, DATA_IN. The data management task would cause the necessary copies to be made and distributed while the calling task waited. Completion of the rendezvous would indicate that the distribution had been completed satisfactorily. When essential data was required (typically for the initialization of a replacement task) the task requiring it would obtain it from the local data management task by calling another entry, DATA_OUT. A general form of the body of this task is shown in figure 4.

Summary

In summary, an Ada program that uses the support system described in section V to allow it to tolerate the loss of one or more processors would have the following form:

(1) A main program with a **begin end** section consisting of a single **null** statement.

```
task body DATA_CONTROL_I is
begin
   loop
      select
         accept DATA_IN(DATA : in DATA_KIND);
            -- Receive data from local tasks, distribute data to
            -- other data management tasks using two-phase commit.
      or
         accept DATA_OUT(DATA : out DATA_KIND);
            -- Provide data to local tasks.
      end select;
   end loop;
end DATA_CONTROL_I;
```

Fig. 4. Body Of Task DATA_CONTROL_I.

(2) A set of tasks providing the various application services; the distribution of the tasks being controlled by an implementation-defined pragma or address clause. Each task would contain handlers for exceptions (such as TASKING_ERROR) that might be generated by the support system if failure occurred while that task was engaged in communication with a task on a remote machine.

(3) A set of tasks designed to provide any replacement service that the programmer chooses; each replacement task suspended on an **accept** statement that will be called to start it executing.

(4) One or more tasks on each processor designed to cope with reconfiguration on that processor; these tasks contain an entry with a parameter that indicates which hardware component has failed. These entries would be called automatically by the support system following failure detection.

(5) A task on each processor designed to distribute copies of essential data for tasks on that processor. Rendezvous with this task allows any other task on the processor to distribute essential data at any time.

VII IMPLEMENTATION

An important question that arises is the execution-time overhead that this scheme introduces. This overhead is determined by the particular data structure chosen for the log in any implementation, and the message volume generated by any particular application. We will use an example data structure in this section for the purposes of discussion. We do not claim that it is in any sense optimal.

The purpose of the message log is to implement failure semantics for Ada. However, other data structures will exist to implement distribution semantics and other execution-time facilities. In particular, a table describing the mapping of tasks to machines will be required. We will refer to this as the task/machine map. This map need not be stored in its entirety on each machine. Only that part needed by the tasks on a particular machine need be stored on that machine.

A simple implementation of the message log could consist of two two-dimensional arrays. Each array is indexed in one dimension by the various machines in the system, and by the local tasks of the machine that the log resides on in the other. One of these arrays, OUT, will be used to record information on outgoing messages that need a reply, and the other, IN, to record information on incoming messages that request entry calls.

An element of OUT is a list. It describes messages that a single local task has sent to a particular remote machine and for which that task expects a reply. When a message is generated it will contain a *message descriptor*, part of which describes the purpose of message. For messages that require a reply, the descriptor will also contain an *identification number*. A reply to a message will contain this same identification number as part of its descriptor. The message log can use the identification number to associate a message with its reply. Thus the only information that needs to be stored in the message list constituting an element of OUT is the descriptor of each message. The message list contains only the descriptors of messages sent to a particular remote machine for which a particular local task is awaiting replies. A given implementation may make it possible for a task to wait for two or more replies. However the maximum number of replies that it is possible to wait for would normally be very small. Apart from this implementation-induced waiting, the only occasion in Ada where a task can wait for more than a single reply is when one task is creating others and awaiting their elaborations.

An element of IN is also a list. It describes the entry calls received for a single local task from tasks on a particular remote machine. The list need only contain the

identification of the caller. A consequence of the rules of Ada is that any task can be on only one entry queue at a time thus the caller's identity identifies the call uniquely [1] The maximum length of a list in IN is the number of tasks in the program, and the average length normally will be much less.

There are three major operations that must be performed on the message log:

(1) Processing a message as it is transmitted.

(2) Processing a message as it is received.

(3) Processing the log when a failure is detected to generate the fake messages.

We will discuss each of these in turn.

Outgoing Messages

The destination of an outgoing message must first be checked against the task/machine map to see if the message is being sent to a machine that has already failed. If so, the descriptor of the message will be used to determine the appropriate fake message to send to the local task.

If the destination machine has not failed then the message descriptor is checked to see whether the message requires a reply. If it does an identification number is added to the descriptor and the descriptor is entered into OUT.

If the message is a reply to an entry call then the corresponding list element in IN has to be removed. The Ada rule stating that entries must be processed in order of arrival ensures that the element to be removed will be the first on the list so that it can be found immediately.

Incoming Messages

If an incoming message requests an entry call then the identification of the sender must be stored in IN. If a message is a reply then the list entry describing the original message is removed from OUT. The list of descriptors constituting the element of OUT

will have to be scanned.

On Failure

When a machine fails the machine/task table must be updated. The row of IN corresponding to the failed machine must be used to remove the corresponding elements in the appropriate entry queues. The row of OUT corresponding to the failed machine must be used to determine which fakes message to send. Finally, the fake messages must be sent to the appropriate local tasks.

Consistent Data

The overhead associated with consistent data depends strongly on the application. The only data that must be kept consistent for all applications, is the machine/task table. This table will need to be updated when a machine fails and when tasks are created, are aborted or terminate.

Normally, an application will need to keep consistent data available across machines so that continued service can be provided. The run-time cost depends both on the amount of data that must be kept consistent and the frequency of updates. Both of these factors can be controlled by the programmer. We expect that for many applications the most recent version of much of the data will not be needed for reconfiguration thereby reducing the potential overhead.

Heartbeats

The run-time overhead associated with heartbeats should be small. If broadcast messages are possible then n messages per interval would be needed. If broadcast messages are not possible, then n(n-1) messages per interval would be needed. Messages could be piggybacked onto existing messages requiring extra messages only when the normal message traffic between machines was not frequent enough. In any case n-1 incoming heartbeats would need to be processed by each machine per interval, however the processing of a heartbeat could be very fast as long as there is no machine failure.

Execution-Time Performance

The overhead associated with processing an incoming or outgoing message involves manipulating the local message log. This overhead is very small compared to the cost of actually sending a message. It will be dependent on the size of the message log. This size will be bounded by:

(max no. local tasks) * (no. machines)
* (max no. of replies a single task can wait for + max no. tasks)

and will usually be much less than this upper bound.

Upon machine failure, fake messages must be sent to local tasks requiring them. No message passing between machines is involved and the situation will occur only rarely. The overhead incurred is not important provided it is bounded and can be achieved within any real-time constraints imposed by a particular application.

The overhead discussed above does not involve any message passing between machines. However, in order to maintain consistent data across all the machines, it is necessary to send many messages. Thus there will be considerable overhead associated with maintaining consistent data. Recall that it is not just data from the application that must be kept consistent. The overhead associated with maintaining consistent data will add considerably to the cost of creating and terminating tasks since the message logs on all the machines must agree. This overhead cannot be avoided if the system is not to be centralized. The application-specific data that must be kept consistent will depend on the application. Again, the overhead will be considerable, but the amount of data and the frequency with which it is updated is under the programmer's control, and so can be kept to the minimum necessary for the application.

A Testbed

An implementation of the tasking, exception handling, and some of the sequential features of Ada incorporating the ideas described in this paper has been undertaken. The goal of the implementation is to demonstrate feasibility and to allow systematic, repeatable demonstrations of the recovery mechanisms. Thus it takes the form of a *testbed* that is heavily instrumented and allows extensive control of the Ada program under test. Simple

programs written with the structure described in section VI have been executed on the testbed and have survived arbitrary processor failures. It must be noted that, of necessity, programs running on the testbed do not execute rapidly.

The testbed provides an arbitrary number of abstract machines and interprets their instruction sets. Communication between the abstract machines is provided by a message passing mechanism implemented by the testbed. Each abstract machine contains a message log as described above and can support any number of Ada tasks. Extensive control and monitoring facilities are provided that allow, for example, Ada tasks to be started and stopped and abstract machines to be failed deliberately. Thus an Ada program can be brought to any achievable state by controlling the order of execution of the tasks, and any combination of abstract machines failed with the program in that state. The program's recovery can then be observed.

A translator translates Ada programs into abstract machine instructions. The testbed executes on a single VAX 11/780 or on a network of Apollo DN300 workstations. Further details of the testbed can be found in [14].

VIII CONCLUSION

Although modern fault-tolerant processors are extremely reliable, they may still fail through degradation or physical damage. In order to benefit from the flexibility of distributed processing, crucial systems must be able to deal with processor failures. In particular, when processor failures preclude the continuation of full service it should be possible to use the remaining processors to provide degraded service. As degraded service can only be specified by the programmer, the programmer must be concerned with both distribution and continuation. This puts certain requirements on any programming language used to program crucial applications for embedded systems.

We have defined two sets of such requirements; distribution semantics and failure semantics. Any language to be used for programming non-transparent continuation must have complete distribution semantics and complete failure semantics. Further, system support to detect failure and provide consistent data must be provided.

Ada was designed for the programming of embedded systems, many of which are crucial and distributed. We have examined Ada's suitability for programming distributed systems in which processor failure has to be tolerated and found it to be inadequate.

Many of the difficulties arise because Ada has no distribution semantics and no failure semantics. We have shown, however, that it is possible to define adequate distribution and failure semantics for Ada, so that the language can be used to program fault-tolerant distributed systems. The new semantics require no additions or changes to Ada's syntax although, as we have described, an extensive support system is necessary to implement them.

The semantics we arrived at were chosen to fit an existing language, and, although they allow fault tolerance to be achieved, they are not ideal. It is clear that distribution semantics and failure semantics should be incorporated into a language at an early stage in its design.

REFERENCES

(1) Reference Manual For The Ada Programming Language, U.S. Department of Defense, 1983.

(2) J.H. Wensley, et al, "SIFT, The Design and Analysis of a Fault-Tolerant Computer for Aircraft Control", *Proceedings of the IEEE*, Vol. 66, pp. 1240-1254, October 1978.

(3) A.L. Hopkins, et al, "FTMP - A Highly Reliable Fault-Tolerant Multiprocessor For Aircraft", *Proceedings of the IEEE*, Vol. 66, pp. 1221-1239, October 1978.

(4) A.S. Tanenbaum, "Network Protocols", *ACM Computing Surveys*, Vol. 13, pp. 453-489, December 1981.

(5) R.D. Schlichting and F.B. Schneider, "Fail-Stop Processors: An Approach To Designing Fault-Tolerant Computing Systems", *ACM Transactions On Computer Systems*, Vol. 1, pp.222-238, August 1983.

(6) D. Cornhill, "A Survivable Distributed Computing System For Embedded Applications Programs Written In Ada", *ACM Ada Letters*, Vol. 3, pp. 79-87, December 1983.

(7) D. Cornhill, "Four Approaches To Partitioning Ada Programs For Execution On Distributed Targets", *Proceedings of the 1984 IEEE Computer Society 1984 Conference on Ada Applications and Environments*, St. Paul, Minnesota, October 1984.

(8) N.G. Leveson and P.R. Harvey, "Analyzing Software Safety", *IEEE Transactions On Software Engineering*, Vol. SE-9, pp. 569-579, September 1983.

(9) Department Of Defense Requirements For High-Order Computer Programming Languages - STEELMAN, U.S. Department of Defense, 1978.

(10) A.J. Wellings, G.M. Tomlinson, D. Keeffe, and I.C. Wand, "Communication Between Ada Programs", *Proceedings of the 1984 IEEE Computer Society 1984 Conference on Ada Applications and Environments*, St. Paul, Minnesota, October 1984.

(11) P.F. Reynolds, J.C. Knight, J.I.A. Urquhart, "The Implementation and Use of Ada On Distributed Systems With High Reliability Requirements", Final Report on NASA Grant No. NAG-1-260, NASA Langley Research Center, Hampton, Va.

(12) P.A. Alsberg and J.D. Day, "A Principle For Resilient Sharing Of Distributed Resources", *Proceedings Of The International Conference On Software Engineering*, San Francisco, October 1976.

(13) J.N. Gray, "Notes On Database Operating Systems", in *Operating Systems: An Advanced Course*, Springer-Verlag, New York 1978.

(14) J.C. Knight and S.T. Gregory, "A Testbed for Evaluating Fault-Tolerant Distributed Systems", Digest of Papers FTCS-14: *Fourteenth Annual Symposium on Fault-Tolerant Computing*, June 1984, Orlando, FL.

DISTRIBUTION LIST

Copy No.	
1 - 3	National Aeronautics and Space Administration Langley Research Center Hampton, Virginia 23665 Attention: Mr. Edmond H. Senn ACD, MS 125
4 - 5*	NASA Scientific and Technical Information Facility P.O. Box 8757 Baltimore/Washington International Airport Baltimore, MD 21240
6 - 7	J. C. Knight
8	A. Catlin
9 - 10	E. H. Pancake/Clark Hall
11	SEAS Publications Files

* One reproducible copy

JO# 7260:lsl

UNIVERSITY OF VIRGINIA
School of Engineering and Applied Science

The University of Virginia's School of Engineering and Applied Science has an undergraduate enrollment of approximately 1,500 students with a graduate enrollment of approximately 560. There are 150 faculty members, a majority of whom conduct research in addition to teaching.

Research is a vital part of the educational program and interests parallel academic specialties. These range from the classical engineering disciplines of Chemical, Civil, Electrical, and Mechanical and Aerospace to newer, more specialized fields of Biomedical Engineering, Systems Engineering, Materials Science, Nuclear Engineering and Engineering Physics; Applied Mathematics and Computer Science. Within these disciplines there are well equipped laboratories for conducting highly specialized research. All departments offer the doctorate; Biomedical and Materials Science grant only graduate degrees. In addition, courses in the humanities are offered within the School.

The University of Virginia (which includes approximately 2,000 faculty and a total of full-time student enrollment of about 16,400), also offers professional degrees under the schools of Architecture, Law, Medicine, Nursing, Commerce, Business Administration, and Education. In addition, the College of Arts and Sciences houses departments of Mathematics, Physics, Chemistry and others relevant to the engineering research program. The School of Engineering and Applied Science is an integral part of this University community which provides opportunities for interdisciplinary work in pursuit of the basic goals of education, research, and public service.